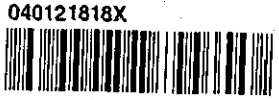


EFFECTIVE INTERPROCESS COMMUNICATION (IPC) IN A REAL-TIME TRANSPUTER NETWORK

By
Mehmet Bor
March 1994

DECLARATION

I declare that the following thesis is a record of research work carried out by me, and that the thesis is my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a degree.

Mehmet Bor

EFFECTIVE INTERPROCESS COMMUNICATION (IPC) IN A REAL-TIME TRANSPUTER NETWORK

By

Mehmet Bor

A Doctoral Thesis

Submitted in partial fulfillment of the requirements
for the award of

Doctor of Philosophy of the Loughborough University of Technology

March, 1994.

Supervisor : *Dr. Ian A. Newman,* PhD.
Department of Computer Studies.

ABSTRACT

The thesis describes the design and implementation of an interprocess communication (IPC) mechanism within a real-time distributed operating system kernel (RT-DOS) which is designed for a Transputer based network. The requirements of real-time operating systems are examined and existing design and implementation strategies are described. Particular attention is paid to one of the object-oriented techniques although it is concluded that these techniques are not feasible for the chosen implementation platform. Studies of a number of existing operating systems are reported. The choices for various aspects of operating system design and their influence on the IPC mechanism to be used are elucidated. The actual design choices are related to the real-time requirements and the implementation that has been adopted is described.

The IPC mechanism exploits the predefined multi-loop network topology of the RT-DOS effectively. It provides a circuit switching message communication service which can be used between any source and destination nodes regardless of their physical location on the network. A number of group communication protocols also are supported, such as broadcasting, multicasting, and domain casting which are all based on an unreliable datagram service. The IPC mechanism was designed with these group communication services in mind at the very beginning, and despite the physical shortcomings of Transputer hardware to supporting group communication, these services have been implemented with a minimum overhead using the lowest level mechanism of the IPC. The IPC provides the processes on the network with location transparency and predictable communication facilities.

KEYWORDS : *Real-time systems, distributed operating systems, object-oriented operating systems, interprocess communication (IPC), group communication, synchronous communication, unreliable datagram services, layered communication services, real-time operating system kernel, predictability, point-to-point connection protocol, dynamic link switching, message switching, circuit switching.*

ACKNOWLEDGEMENTS

I thank *Dr.I.A.Newman* for his directions, valuable suggestions, and corrections he made for this thesis. I owe him my gratitude for his moral support and deep trust in me, which have been sources of motivation and encouragement through this research work. I also thank P*rof. I.Edmonds* for his interest in my research, and the *Science Engineering Research Council* for the provision of a research studentship for me.

I also thank D*r.Murat Tayli*, *Dr.M.Benmaiza*, and *Mr.Bradley Swim* for their valuable criticisms, suggestions and encouragement during the implementation.

Lastly, I am indepted to my wife *Raside* for her patience and understanding through all the past four years full of many busy weekends.

Mehmet Bor
March, 1994

CONTENTS

1. *INTRODUCTION*

Though there have been many real-time operating systems available for dedicated systems in the industrial field, none of them has yet distinguished itself as a general solution to this class of problems. The main reason for this is the stringent set of requirements of real-time applications which are quite different from the business or scientific applications [TayliA 1987].

The primary characteristic of real-time operating systems is their ability to function under time constraints imposed by the physical process being monitored and controlled. Because of the dynamic nature of the external processes, the scheduling and dispatching functions of the operating system should follow a dynamic and adaptive strategy. In large number of cases, it is not feasible either for safety or for economical reasons, to stop or to shut down an entire control system in order to change, repair, maintain or extend parts of it. The operating system should provide the dynamic reconfiguration support to increase the reliability and the extendibility of the overall system. Although fault detection is mainly a job of the hardware, it is the role of system software to recover the system from failure after the fault is detected and to ensure the system's continuity.

Distributed computer systems seem to provide adequate architectural characteristics and suitable mechanisms to satisfy the above requirements [Enslow 1978, StankovicA 1984].

The specification and implementation of this class of systems is also a critical issue, where the designers select generally one of the two opposite approaches : the virtual machine abstraction or the machine tailored solution. The virtual machine approach, which hides machine dependent details, seems to ensure better portability but often fails to meet the performance criteria. On the other hand, machine dependent approach requires the duplication of system design and implementation effort, causing non-negligible time and money consumption.

The new trend in this area seems to be the search for an adequate trade-off between these extremes. Japan's TRON project [Sakamura 1987], in which more than 50 companies, mainly Japanese, and academies from a number of countries have participated, has

provided an important step in the field. The idea is to provide a design of open system computer architecture, very similar to the open system interconnection model of ISO which defines communication protocols in the frame of protocol layers [Zimmerman 1980], which will specify the entire computer system in terms of well defined logical layers. Consequently, an industry oriented real-time operating system can be specified and designed as a series of machine independent layers, all based on a single architecture dependent interface.

The work presented in this thesis stemmed from a research effort of the specification, design and validation of a distributed operating system kernel which will support industrial applications : the RT-DOS Kernel [TayliA 1987, TayliB 1990]. The subject of the thesis is the design and implementation of an interprocess communication (IPC) mechanism for this kernel. Though the IPC mechanism is a self contained part of the kernel, it is interrelated to all other kernel mechanisms, such as naming and resource management, and hence the work that has been done which is related to these subjects is summarized in the thesis.

1.1 General Background of Real-Time (RT) Distributed Systems

Traditionally, application domains of Real-Time systems can be found in process control, command and control systems and embedded systems [Benmaiza 1990]. Such systems are generally characterized by a static view of the controlled environment. The behavior of these "static" systems can be therefore preplanned. Since formal proofs of time constrained systems are still in the early stages of research, implementation of such systems requires a heavy phase of simulation in order to gain necessary confidence in their correct time-dependent operation. This static approach results in building non-flexible systems. A single change in the system may require a complete, lengthy and costly simulation phase to ascertain that the system is respecting the specified time constraints. The necessity of using simulation techniques to verify the time correctness of a given system stems in fact from its unpredictable timing behavior. It is very difficult, for example, to know in advance how long a task will be waiting for the availability of a given resource. Note that simulation-based approaches put a formidable responsibility on the designer who, somehow, has to force manually the system under design to be predictable by performing changes in the code of the tasks. A better

approach would be to put parts of the predictability verification process into the operating system.

Unpredictability can come from three different sources: unpredictable resource allocation policies, unpredictable execution times of asynchronous events, and unpredictable communication delays. The objective of the thesis is to provide the upper operating services with a predictable interprocess communication mechanism, so that the operating system can establish a predictable resource allocation schema at the processes level.

1.2 Background and Scope of the RT-DOS Project

Since 1986 at King Saud University of Riyadh/Saudi Arabia, a research team has been investigating the design and implementation of a Real-Time Distributed Operating System (RT-DOS) based on Transputer networks [TayliA 1987, TayliB 1990, BorA 1989, TayliC 1986, TayliD 1990, Aytac 1992, BorB 1990, Benmaiza 1990, TayliE 1990]. The first phase was dedicated to the study of existing systems, identification of the current and future trends in Real-Time computing and the definition of clear objectives for the design and implementation of the projected system. The first step resulted in the definition of a kernel providing a unique system abstraction hiding the physical distributed nature of the underlying system. The kernel, replicated at each node, provides the following basic services [TayliB 1990] :

- Process management;
- Memory management;
- Interprocess communication (IPC);
- Time management;
- Short-term scheduling; and
- Process migration.

The preliminary study of the kernel has clearly shown that its functions can and should be separated into two sublayers based on the desirable separation between mechanisms and policies. A first sublayer implements necessary mechanisms needed to support a second policy-oriented sublayer. In the design of the kernel the first mechanism-oriented sublayer is seen as "domain-independent" in the sense that it implements mechanisms common to various types of operating systems. The second sublayer is considered as "domain specific" and as such, implements intrinsic features of the target system, namely

real-time and distributed features. This approach enforces the view that the main difference between various categories of Operating Systems resides much more in the policy-making parts (particularly in the resource scheduling) than in the mechanisms necessary to implement these policies. The separation between policies and mechanisms is known to be important in integrating dynamic aspects of modern Real-Time systems [HideyukiA 1989, StankovicB 1989, StankovicC 1985].

The hardware equipment and software tools which have been used in the above research project are as the following:

- **Hardware Equipment:** IMS B006 Board with a T212 Transputer and RS232C Serial Port on it; IMS B004 PC Development Board with a T414 Chip, a C012 PC Link, and 2MB on-chip RAM; TMB12 Board with two IMS *C004* Programmable Link Switches controlled by a T212 Transputer, one T805 TRAM Module, and ten T425 TRAM Modules (with 1MB RAM each) on it; B003 Board with four T414 Transputer chips (each has 256 KB on-chip RAM).

- **Software Tools:** OCCAM 2 and TDS (Transputer Development System) Toolset for IBM PC Environments (IMS D700 D), IMS D7214 Transputer Development System with ANSI C Based Toolset.

Though at the beginning of the project the OCCAM Language [Geraint 1987, Pountain 1987] based TDS D700 D Toolset [InmosF 1988] had been used for about three years, because of the some shortcomings and limitations of OCCAM language, all the OCCAM programs have been converted into Parallel C language which is available with an ANSI C based IMS D7214 development environment [InmosG 1990]. The reason for such a radical change was that, though OCCAM is a simple but yet powerful concurrent programming language based on the Hoare's CSP definition [Hoare 1978] which lends itself well to application software development on Transputers [Transputer 1984] for embedded systems, it doesn't provide enough flexibility for operating system (especially low level kernel mechanisms) design and implementation. It lacks a number of features that are very crucial for operating system implementation, such as dynamic memory allocation, I/O handling, asynchronous communication, etc. It is hiding some hardware features of Transputers from programmers. OCCAM TDS D700D also has some restrictions regarding the dynamic configuration of OCCAM programs on Transputer

networks, and the generation of relocatable code for Transputers (which is necessary for task migration to implement dynamic load balancing and fault tolerance concepts). The ANSI C Toolset (IMS D7214) removes most of all these limitations and (except for the limitations of Transputer hardware architecture itself) provides more freedom and flexibility.

1.3 Importance and Role of the IPC Mechanism in the RT-DOS Project

The RT-DOS Kernel, replicated at each Transputer node of a network of Transputers, provides the following basic mechanisms : process management, memory management, time-management, short-term scheduling, and interprocess communication (IPC) mechanisms. The IPC mechanism is at the heart of the RT-DOS Kernel, and its correct design is vital to the efficient working of the systems designed on top of it. It provides various types of communication, starting at the lowest level datagram service, up through exactly once delivery, to bi-directional communication and RPC support. To support predictability feature at the upper levels of the RT-DOS Kernel (for example, task scheduling and resource allocation), the IPC mechanism should avoid unpredictable communication delays, and provide predictable communication services to these services. For any real-time distributed system, the only guarantee of the predictability of the other system services is the timely availability of the communication medium (i.e., resource availability).

As the RT-DOS Kernel is implemented using the server model, replicating basic functions at each kernel and accessing them through the standard IPC interface [CheritonB 1983], implementation of other kernel modules is simplified at the expense of including supportive services and increasing the IPC complexity. Because of the granularity of the RT-DOS Kernel components, the naming issues which are closely related to the naming of the communication elements, are handled by the IPC naming schema.

Finally, as the RT-DOS is a message-based system [TayliB 1990], all concurrency control and process coordination mechanisms were expected to be implemented with the IPC mechanisms.

1.4 Scope, Objectives and Limitations of the Thesis

This thesis is devoted to the design and implementation of an IPC mechanism for the RT-DOS Kernel explained in **Section 1.2**, and **Chapter 3**. Given the physical topology of the underlying Transputers network and its semi-dynamic reconfiguration ability, the synchronous characteristics of the OCCAM programming model were investigated. The objective was to find the most efficient and flexible IPC mechanism for providing the basic services to the upper kernel levels.

The proposed physical model of the RT-DOS confines control messages within concerned application domains, shortens communication paths (with respect to the communication capability of Transputers) and prevents unnecessary overheads on shared transmission media. Moreover, point-to-point data transfer avoids the problems of buffer allocation, communication congestion, and extra copying. The duration of data transfers will be a function of message length, link transfer speed, and the cost of a limited number of control messages establishing the direct path. As a result, the RT-DOS IPC is expected to provide low and predictable communication overheads, much demanded by real-time applications [TayliB 1990].

Though, the main objectives of the thesis were to design primitives, message structures, and protocols of an IPC mechanism for the RT-DOS Kernel, a number of basic decisions were initially specified to delineate the kernel model. These decisions can be summarized as the following :

- The IPC protocols, which are planned to provide connectionless transport level communication;
- The IPC primitives, which will be of blocking type and the concurrency will be supported by the use of lightweight processes;
- Control messages (small and fixed size);
- Typeless data transfer, which will preclude automatic data conversion and control of system capabilities.

Messages are unbuffered and higher level protocols will ensure reliability of communication primitives. The IPC is supposed to handle the problems of implementing a server model RT-DOS kernel; and enable the serialization, queuing, and/or multiplexing of such services. The decision, about whether the complementary

synchronization constructs (semaphores, monitors, etc.) are needed or not, would be based on the results of the design effort of the IPC model.

1.5 A Brief Summary of the Thesis Contents

In the thesis, after a brief introduction of the real-time distributed systems' features (**Section 1.1**), the background and the general scope of the RT-DOS project with the importance and relevance of the IPC mechanism in it, have been submitted in different sections of **Chapter 1**. In **Chapter 2**, operating system concepts and terminology in general, as well as real-time system requirements and distributed system characteristics are defined in its different sections.

After introducing the general operating system concepts, the RT-DOS Kernel project and its objectives are also explained (**Section 3.1**), summarizing the relevance of the IPC mechanism with the research project, and its relative importance. Since the RT-DOS Kernel and its IPC mechanism are implemented on Transputer network platforms, the hardware architecture of Transputers and related software development tools (including the currently used project test-bed equipment) with their basic capabilities and limitations, are introduced (**Section 3.2**). At the end of the chapter, different design approaches to the RT-DOS Kernel (**Section 3.4**), and the survey study carried out about the related works which have been done in the past, are submitted (**Section 3.5**).

In **Chapter 4**, the current RT-DOS architecture with its logical and physical models (**Sections 4.7 and 4.8**), as well as its basic design considerations (i.e., object naming, resource management, concurrency control, and interprocess communication schema), are presented.

The IPC implementation work which has been the main topic of the thesis, has been presented in **Chapter 5**, in detail. In this chapter, a brief explanation of the current testbed, with the elaboration of the important hardware components which are involved in the IPC implementation, are submitted (**Sections 5.4.1, 5.4.2, and 5.4.3**). A top to down approach is used in explaining the IPC architecture, starting from the basic communication components (**Section 5.3**); then, submitting network topology, the IPC layers (**Section 5.4.4**), the communication primitives (**Section 5.4.5**), datagram messages and message exchange protocols (**Section 5.4.6**); and finally, giving the basic

data structures used in carrying out the message exchange between communicating parties (**Section 5.4.7**).

An elaboration of other possible design approaches to the IPC implementation is given in **Chapter 6**. In the conclusion chapter (**Chapter 7**), the basic lessons learned from the project during the research study, with a brief summary of the achievements of the current work which has been done, in comparison with the objectives which were set initially, are presented. In **Chapter 8**, the areas of further research which have been excluded from the current project context, are listed. The list of the references of articles, and figures which are presented in different sections, are submitted at the end of the thesis.

2. GENERAL CONSIDERATIONS AND TERMINOLOGY

In this chapter, the basic operating system concepts, as well as the most important features of real-time and distributed systems that distinguish such systems from the others (such as parallel processing systems, tightly/loosely coupled systems, and network operating systems) are presented. Moreover, the main role of an IPC mechanism in real-time DOS kernels in general, and expected functions of the designed IPC mechanism in the RT-DOS Kernel in particular, are briefly summarized.

2.1 Operating Systems (OS) and Distributed Operating Systems (DOS)

An operating system (OS) may be viewed as an organized collection of software extensions of hardware, consisting of control routines for operating a computer and for providing an environment for the execution of programs [Milonkovic 1992]. Other programs usually invoke services of the operating system by means of operating-system calls. It acts as interface between users and the hardware of a computer system. Internally, an operating system is a manager of resources of the computer system, such as processor, memory, files, and I/O devices. The range and extent of services provided by an operating system depend on a number of factors; and, user-visible functions of it are determined by the needs and characteristics of the target environment that the OS is intending to support.

With regard to the aspects of processor scheduling, memory management, I/O management, and file management, operating systems can be categorized into batch, multiprogramming (or multitasking), real-time, distributed, and combination operating systems (such as real-time distributed operating system). As the subject of the thesis is directly related to the real-time distributed operating systems which is attributed to the objectives of the RT-DOS Kernel project, the other types of operating systems are excluded from the thesis.

Distributed Systems : Distributed systems is a generic name for a number of decentralized computer systems architectures. Most research and many realizations have been reported in the literature on the issue. A number of researchers identified the problem in different ways [Enslow 1978, StankovicA 1984, Eckhouse 1987, StankovicD 1985, Turek 1992, Hariri 1992]. In the context of this research, a distributed system will be defined as a decentralized architecture, consisting of a multiplicity of physically

dispersed processing nodes, integrated into a single and transparent system through a native, global decentralized operating system. A decentralized operating system manages the system's collective, disjoint physical and logical resources to present the users a unified and transparent view of the system.

Distributed computer systems and distributed processing concepts refer to a relatively new kind of computer architectures and devices which have progressively evolved over the last decades, due to technological changes in micro-electronics, communications, and ever growing user needs. If we try to characterize a distributed system, as opposed to a centralized system, it will be seen that definition boundaries are confusing and not clear. Most computing systems include such basic components as hardware, system data, system software, user data, and user software. Any computer system can be named as a distributed computer system, if it has some (or all) of the above components distributed [TayliC 1986]. As a matter of fact, the technical and commercial literature are full of contradictory and overlapping definitions. The developments in the networking and inter-networking, and standardization trends made the integration of geographically distributed heterogeneous computer systems possible and simple. Well known standards at all levels, i.e., RS232, HDLC, X.25, SNA, DECNET, etc. [TanenbaumE 1980], offer a fair level of interconnection possibilities. The OSI model developed within ISO [Zimmerman 1980] is intended to decrease the number of existing incompatibilities in the future.

The goals motivating most computer system development projects which stemmed generally from managerial and economical considerations, are the following:

- Increased system productivity (greater capability, shorter response time, and increased throughput);
- Improved reliability and availability;
- Ease of system expansion and enhancement;
- Graceful growth and degradation; and
- Improved ability to share resources.

Each new major systems concepts or development, e.g., multiprogramming, multiprocessing, networking, etc., has been presented as the answer to achieve all of the goals, such as high speed, high capacity, reliability, modular growth, availability, and adaptability and many others. The following is the list of some of the benefits currently

claimed for distributed computer systems in the recent commercial and academic literature:

- High availability and reliability;
- Reduced network costs;
- High system performance;
- Fast response time;
- High throughput;
- Graceful degradation, fail-soft, error-containment, fault-tolerance;
- Ease of modular and incremental growth;
- Dynamic reconfiguration;
- Automatic load balancing and resource sharing;
- Easily adaptable to workload changes;
- Incremental growth in capacity and/or function (replacement and/or upgrading);
- Good response to temporary overloads and exceptional situations.

Many multi-machine and/or multi-processor systems may present similar characteristics to the definition above, yet they differ in fundamental design and goal issues. For example, computer networks are often said to be distributed systems. The main difference between a computer network operating system (NOS) and a truly distributed operating system (DOS) is that a NOS is a collection of software layers added on top of connected machines' local operating systems on the network, while a DOS is a native operating system for all the distributed nodes which is designed with the networking requirements in mind from inception [Kimbleton 1978]. All of these concepts are briefly summarized below. A very detailed literature survey on distributed computer systems concepts has been carried out by the RT-DOS Kernel project team, and the results have been presented as a report [TayliC 1986].

Network Operating Systems (NOS) : In a NOS environment, each host of a computer network has a local operating system that is independent of the network (even when they are replicated!). The sum of all operating system software added to each host in order to communicate and share resources is called a NOS. The added software often includes modifications to the local operating systems. NOSs are characterized by being built on top of existing OSs, and they attempt to hide the differences between the underlying computers. The most famous of such computer networks is ARPANET [McQuillan 1977] and it contains several NOSs, such as RSEXEC, NSW, and XNOS.

Although the above mentioned NOSs provide the concept of physical and logical resource sharing, they distinguish themselves from distributed system family, with respect to the following characteristics:

- Each computer has its own OS, following a local policy, instead of running as part of a global, system wide OS;
- Each user normally works on his own machine and has to use specific remote login procedures to share the resources, instead of having the OS dynamically allocate them;
- Users are typically aware of where their files are kept and must move files between machines with explicit file transfer commands, instead of having this managed by the OS;
- Users refer to remote resources by the name of the resources rather than by the name of the service which is attached. That is against the idea of transparency and fault tolerance;
- The system has little or no fault tolerance; if 1 percent of the system crashes, 1 percent of the users are stopped, instead of everyone simply being able to continue normal work, albeit with some percent of worse performance.

Network operating systems are out of the context of the RT-DOS Kernel project, and are excluded from the thesis as well, intentionally. The objective of the work described in the thesis is to implement a message-based IPC mechanism on multi-computer systems (network of Transputers) which do not have shared main memory, separating it from the research area performed on multiprocessor hardware platforms in which a common memory is shared between the nodes [Jones 1979, Ousterhout 1980, Wulf 1981].

Distributed Operating Systems: A distributed operating system (DOS) is designed with the networking requirements in mind from its inception. It is the only native OS for all the distributed hosts. The main characteristic which influences the design and architecture of DOSs is the lack of consistent and up-to-date information about the global status of the system, as opposed to a centralized OS where executive functions can gather almost instantaneously whatever data is available on the working context. It is generally not a good idea in the distributed environment to even try to collect complete information about any aspect of the system in one place. Some components may answer with noticeable delays or may not answer at all. This approximated view of the system

makes any of the centralized systems' algorithms impossible or impractical in the decentralized context. However, the lack of a central locus of control is the major strength of a decentralized system where the failure of such component would be fatal for the overall system.

The main issues to be considered in the specification and design of a DOS are:

- Interprocess communication (IPC) primitives which is the subject of this thesis;
- Naming and protection of system objects;
- Resource management;
- Fault tolerance and reliability;
- Basic services for real-time operations.

Parallel Processing Systems: An important theme of system development has been parallel processing, in which a series of processors, general or specific purpose, are coupled to carry out a given task. System coupling refers to the means by which two or more processing units exchange information. The coupling relates to physical data transfer as the manner in which the recipient of the data responds to its contents. These two aspects of system interconnection are called physical coupling and logical coupling, and they are present in all multiple component systems whether the components of interest are complete computer systems or smaller assemblies.

The terms **tight** and **loose** have been utilized to describe the mode of operation for each type of coupling. The interconnection and the interaction of two computer systems can then be described by specifying the nature of its logical coupling. It is important to point out that all four combinations of these characteristics are possible and that they all have been observed in implemented systems.

Tightly-coupled systems are characterized by the use of a shared memory as the communication media, among multiple processing units. The processing units may be general purpose processors or special purpose processors, like array processors, arithmetic logic units, etc. All these processors may have their own local memories, but they all communicate through a common address space. Most of these multiprocessor systems also adopt a tight logical coupling policy. Consequently, the recipient of the message is required to perform whatever service is asked by the sender. The relation among them is basically that of master-slave. Although the concept of tightly-coupled

multiprocessor systems appears to be a viable approach for achieving almost unlimited improvements in performance (i.e., increased system throughput) with the addition of more processors, this has not been the result obtained with the implemented systems.

It is the very nature of tight-coupling that results in limitations on the improvements achievable. Some of the ways that those limitations have manifested themselves are listed below:

- The direct sharing of resources (memory and I/O) often results in access conflicts and delays in obtaining use of the shared resources;
- Programming languages that support the effective utilization of tightly-coupled systems have not been developed;
- The development of optimal scheduling for the utilization of the processors is very difficult, except in trivial and static cases;
- Any inefficiencies present in the OS appear to be greatly exaggerated in such systems.

The front-end computers, largely used in batch systems to carry out slow I/O operations, constitute physically loosely-coupled systems, but logically they are tightly-coupled systems. Later on, specialized vector and array processors, and specialized units, like Fourrier transformers, were connected to general purpose computing environment and utilized as attached support processors. In any case, the specialized nature of the services provided by these attached processors exclude them from consideration as possible approaches to provide general purpose support.

For a more detailed elaboration of different aspects of tightly-coupled shared-memory systems please refer to [Woodward 1981].

Loosely-coupled systems are multiple computer systems in which individual processors both communicate physically and interact logically with the others by exchanging messages through communication channels at the input/output level. There is no direct sharing of primary memory, although there may be sharing of secondary storage devices, printers, etc. The unit of data transferred is whatever is permissible on the I/O path being used. In order to achieve a data transfer, the active cooperation of both processors is required during the communication process.

The most important characteristic of loosely logical coupling is that one processor does not have the capability or authority to force the other processor to do something. One processor can deliver data to another; however, even if the data is a request for a service to be performed, the receiving processor, theoretically, has the full and autonomous rights to refuse to fulfill that request. The reaction of processors to such a request for services is established by the operating system rules of the receiving processors, not by the sender. This allows the recipient of a request to take into consideration local conditions in making the decision as to what actions to take.

It is important to note that a system can be physically loosely-coupled but logically tightly-coupled and their relation may be a master-slave relation. The RT-DOS Kernel is designed to run on every node of a physically loosely-coupled network of Transputers which interact with each other in loosely-coupled manner to accomplish required upper level services.

2.2 Fundamental Concepts of Real-Time (RT) Systems

In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced [StankovicE 1988]. Examples of current real-time computing systems include the control of laboratory experiments, the control of automobile engines, command and control systems, nuclear power plants, process control plants, flight control systems, space shuttle and aircraft avionics, and robotics. Real-time command and control systems play an important role in our daily lives, and as they are the subject of the RT-DOS research project and the IPC mechanism is supposed to be used by the RT-DOS kernel, these systems will be discussed in relatively more detail below.

Real-time systems are characterized by the fact that severe consequences will result if logical as well as timing correctness properties of the system are not satisfied [StankovicF 1988]. Typically, a real-time system consists of a controlling system and a controlled system. For example, in an automated factory, the controlled system is the factory floor with its robots, assembling stations, and the assembled parts, while the controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor. Thus, the controlled system can be viewed as the environment with which the computer interacts. The controlling system interacts with its

environment based on the information available about the environment, say, from various sensors attached to it. It is imperative that the state the environment, as perceived by the controlling system, be consistent with the actual state of the environment. Otherwise, the effects of the controlling system's activities may be disastrous. Hence, periodic monitoring of the environment as well as timely processing of the sensed information is necessary. Timing correctness requirements in a real-time system also arise because of the physical impact of the controlling systems' activities upon its environment. For example, if the computer controlling a robot does not command it to stop or turn on time, the robot might collide with another object on the factory floor.

Timing constraints for tasks can be arbitrarily complicated, but the most common timing constraints for tasks are either **periodic** or **aperiodic**. An aperiodic task has a deadline by which it must finish or start, or it may have a constraint on both start and finish times. In the case of a periodic task, a period might mean "once per period T" or "exactly T units apart". In most of real-time systems, activities that have to occur in a timely fashion coexist with those that are not time-critical. A task with a timeliness requirement is called a real-time or time-critical task. Ideally, the computer should execute time-critical tasks so that each task will meet its timeliness requirement, whereas it should execute the non-time-critical tasks so that the average response time of these tasks is minimized. It is to be noted here that overall throughput of the system in real time systems has a secondary importance. The need to meet the requirements of individual time-critical tasks is one issue that makes the problem of designing a real-time system a hard problem. Other issues include fault tolerance and the need to operate in uncertain environments. Low level application tasks, such as those that process information obtained from sensors, or those that activate elements in the environment, typically have stringent timing constraints dictated by the physical characteristics of the environment. A majority of sensory processing is periodic in nature. An example of a dynamically created task is a (periodic) task that monitors a particular flight; this comes into existence when the aircraft enters an air traffic control region and will cease to exist when the aircraft leaves the region. In addition, time-related requirements may also be specified in indirect terms. For example, a value may be attached to the completion of each task where the value may increase or decrease with time; or a value may be placed on the quality of an answer whereby an inexact but fast answer might be considered

more valuable than a slow but accurate answer. In other situations, missing x deadlines might be tolerated, but missing x+1 deadlines can not be tolerated.

In a static system, the characteristics of the controlled system are assumed to be known a priori, and, hence, the nature of activities and the sequence in which these activities take place can be determined off-line before the system begins operation. Needless to say, such systems are quite inflexible even though they may incur lower runtime overheads.

Real-Time Command and Control Systems: The computer systems which are used to control a collection of physical processes by sensing and altering their states under time constraints, are called real-time command and control systems which are the interest of the thesis. The state of these physical processes change independently as a result of external conditions, which are not completely under the control of the computer system. Any type of command and control system can be roughly divided into three levels:

- Low level synchronous sampled data loop functions (like sensor/actuator feedback control, signal processing, etc.) which require responses within microseconds for external events;
- Middle level supervisory control functions, above the sampled data loop functions, which require responses within milliseconds for any action ranges, such as opening/closing a walve or changing direction of a robot;
- Human interface management functions which require responses within seconds or sometimes minutes, such as displaying system status changes on CRT screens in graph forms.

The type of control referred in the RT-DOS research is the middle level supervisory functions which provide basic primitives to set up the two application dependent surrounding layers. Examples of real-time command and control systems are found in plant (e.g., factory, refinery) automation, vehicle (e.g., airborne, shipboard) control and surveillance (e.g., air traffic control) systems.

Conventional practice in real-time computing systems today is to provide the minimal functionality and to pass the time, space and intellectual complexity borders of system resource management on to the application programmer. These executives strive to avoid doing anything that would make it difficult for applications to meet their time constraints and try to provide service to the clients in a predictable manner. The consequence is the

increase in system's implementation cost and often a degradation in its performance due to the contradictory specifications. The RT-DOS project also focuses on the level of functionality provided by the native system to enhance the programmer's support.

Real-Time Operating Systems, Architecture and Hardware: The key issue related to real-time operating systems is predictability which requires clean operating primitives, some knowledge of the application, proper scheduling algorithms, and a viewpoint based on a team attitude between the operating system and the application. The OS must be able to perform integrated CPU scheduling and resource allocation so that collections of cooperating tasks can obtain the resources they need, at the right time, in order to meet timing constraints. The cooperation requirement means that there is an end-to-end timing requirement (i.e., a collection of activities must occur - with complicated precedence constraints - before some deadline). Using the current OS paradigm of allowing arbitrary waits for resources or events, or treating the operation of a task as a random process, it will probably not be feasible to solve this complicated set of requirements. An important realistic and complicating factor to the integrated resource allocation problem is the need to be predictable in the presence of faults.

Because of the timing properties of real-time systems, specifically the execution times of the various tasks are very tightly related to the underlying hardware, and quite often an early binding of the logical functions to the physical hardware units is done. The point at which this binding is done, in part, determines the adaptability and predictability properties of the system. Depending on the complexity of a controlled system, the static versus dynamic nature of the activities of the environment, the complexity of the tasks executed by the controlling system, and the overall goal of the system, design and implementation strategies adopted will vary.

Hard real-time systems are usually special purpose. Architecture to support such applications tend to be special purpose too. By **hard real-time** tasks, it is meant that tasks must complete their activities by their "hard" deadline times, otherwise it will cause undesirable damage or fatal errors to the system, while in **soft real-time** systems tasks do not have such "hard" deadlines and it still makes sense for the system to complete the tasks even if they passed their "critical" (i.e., soft deadline) times [HideyukiB 1987].

A number of suggested rules which are related to the real-time system development [TayliA 1987, TayliB 1990, Nortcutt 1987, StankovicC 1985, StankovicB 1989, StankovicE 1988] can be listed as the following:

- Develop special purpose configurations of off-the-shelf, general components;
- Do not change the problem to fit the hardware;
- Fault tolerance and real-time capability must be designed in at the outset;
- Growth limitations of the system are strongly influenced by the growth in overhead as modules are added;
- Perform functional partitioning (not very rigid), but avoid too static schedule;
- Provide on line testability;
- Private memories can be used for read-only code, and global memory can contain shared data only;
- Dynamically loaded local memories and caches cause many problems difficult to deal with under timing constraints because they may violate the principle that repeatability of timing is crucial.

Communication in Real-Time Systems: There is a need in real-time systems for primitives such as timed semaphores, timed monitors, real-time datagrams, real-time virtual circuits, stream transactions, and real-time transactions, to coordinate a number of concurrent processes which are running independently. These primitives support the implementation of effective resource sharing mechanisms as well as a number of effective inter-process communication protocols. A real-time datagram might be defined as a datagram that must be successfully delivered by time *t*. Sets of cooperating tasks are the norm for distributed, hard real-time systems. The semantics of the communications varies as well as the interconnection structure between the communicating tasks and their timing requirements. Reliability, which is a prerequisite and the most important requirement for real-time systems, can not be achieved if any of the system components, such as task communication services (IPC) or scheduling policy, is not reliable in terms of predictability.

In the design of distributed real-time systems, there is a need for the use of communication protocols that provide for deterministic behavior of the communicating components. In particular, this requires protocols that result in bounded message communication delays. A number of advances have been made with regard to protocols for time-constrained communication. Though some new access control protocols are

being developed, integration of the low-level protocols with the functioning of the OS kernel, I/O modules, and application modules, as well as the inclusion of fault-tolerance features, are issues that remain to be solved. Fault tolerance must be designed in at the start, must encompass both hardware and software, and must be integrated with timing constraints [StankovicE 1988, Hideyuki 1989].

The type and constraints of real-time applications seem to impose required system configurations. Structuring of real-time applications in terms of parallel activities can be achieved in a number of ways [Hull 1989, Newman 1981, InmosI 1988]. The first approach consists of decomposing the problem into a number of smaller components which can be executed in parallel. The second category contains applications where the parallelism has been obtained by distributing the data to be processed between a number of processors (MIMD architecture). Another group consists of applications where a number of processors are used to process data farmed out by a controlling processor. It would be noted that all these approaches are not mutually exclusive, and a given application can be structured using one or more of these models. Following the decomposition step, the application needs to be verified with respect to its functional specifications, and later on validated using analytical and simulation models. However, specification, verification, and validation of real-time applications are beyond the scope of the RT-DOS project. These issues are addressed in a parallel research sponsored by C.C.I.S. Research Center [Aytac 1992].

Following these off-line activities, real-time applications are installed (configured) on the target system, taking into account their resource and time constraints. An operating system has to provide the necessary tools to monitor their performance and predict timeliness criteria. Mapping of a network of tasks on a network of processors, forming the real-time applications and monitoring them are carried out by:

- Application generators;
- System configurer;
- Resource manager;
- Schedulers; and
- Application managers.

All these activities are carried out by high level OS services (mostly policy making components) which are also beyond the scope of this project. These issues are addressed in a parallel research project sponsored by C.C.I.S. Research Center [TayliF 1989].

2.3 Real-Time Distributed Operating Systems (RT-DOS) and RT-DOS Kernels

Real-time distributed operating systems carry the distinctive characteristics of both real-time systems and distributed systems, such as distributed functionality of system services on physical dispersed nodes and predictability.

The experiences with the design and implementation of centralized and distributed OSs and the state of art in surveyed literature provide us with a number of observations that profoundly influenced our general approach to the definition and design of the RT-DOS Kernel. These points can be summarized as follows [TayliB 1990]:

- Basic functions in real-time systems do not differ conceptually from those found in other operating system classes. The dissimilarities of real-time systems stem from :
  - . different emphasis in their scheduling policies,
  - . the need for better predictability of system behavior,
  - . stringent performance expectations,
  - . their ability to be physically and functionally reconfigurable, in order to address small, or large scale applications.

  The first two points are associated with policy making parts of an OS, while the other two refer to functional specifications, such as granularity, modularity of basic components, and implementation optimizations;
- In distributed systems, a relatively small OS kernel can implement basic protocol and services, providing a simple network transparent process address space and communication model. The rest of the system can be built at process level, in a machine and network independent fashion, supporting policy making components and services required by the intended class of application;
- High performance communication is the most critical facility for a distributed computer system (given the available technology). Thus, the challenge is to

design the protocols that lead to a system with the required performance, functionality, reliability, and some degree of security;

- Fault-tolerance is inherent to the basic architecture of a system, hence operating systems with such objective should provide necessary mechanisms as part of their basic services.

These observations suggest that a carefully designed distributed operating system kernel can support different categories of system, including the real-time class, provided that necessary services are offered with due performance. Therefore, in the design of the RT-DOS Kernel the emphasis is put on the distributed attribute of the system, considering the impact of the real-time factor as either an optimization issue, or the concern of high level system services.

2.4 General OS Kernel Functions and Policy/Mechanism Separation in OS Design

The kernel defined here is intended to be a collection of mechanisms which support a range of system solutions that effectively meet the requirements of various reliable, distributed, real-time command and control applications. The distributed real-time command and control application domain calls for a set of kernel mechanisms that support a number of reliability concepts, such as fault containment, graceful degradation, high availability of services, and correctness of actions. The mechanisms mentioned here do not constitute a full operating system, but rather an operating system kernel which provide fundamental system interfaces that are not the same as a trivial operating system or executive.

The concept of policy/mechanism separation has been claimed to be valuable in the design of modular operating system components [Hansen 1970, Haberman 1970]. Briefly, a policy is defined as a specification of the manner in which a set of resources are managed, and a mechanism is defined as the means by which policies are implemented. Policy/mechanism separation is a structuring methodology that involves the segregation of entities that dictate resource management strategies from entities that implement the low-level tactics of resource management. The main idea behind the policy/mechanism is that if the mechanisms are pure (i.e., devoted to policy decisions)

and complete, then it is possible to use them in implementing a wide range of the system- and application-level facilities.

Modularity is achieved through the separation of functions into mechanisms; implementation changes are restricted to individual mechanisms, and changes in system policy do not require changes in the functionality of mechanisms, just changes in the use of mechanisms. A number of functions necessary in the creation of reliable, distributed, real-time systems are not provided by the kernel to keep its size at a minimum. In these cases, the functions are supported by kernel mechanisms, but the specific policies are applied at higher levels in the system. For example, though the RT-DOS kernel provides a process migration mechanism, the function of dynamic reconfiguration is to be performed by system-level facilities that manage the physical location of objects (server processes, etc.).

During the initial phase of the RT-DOS Kernel design, a great deal of effort was spent to define a micro kernel which consists of a collection of basic mechanisms duplicated at each node of a physically loosely-coupled network of Transputers, with a minimum functionality and maximum flexibility. The rest of the kernel mechanisms, as well as other policy level OS servers, are relying on the interprocess communication mechanism (IPC) to communicate among themselves by exchanging messages.

The RT-DOS Kernel is implemented as a collection of kernel-level mechanisms (IPC, memory management, process management, short-term scheduling, time management, and process migration) from which policy decisions were carefully excluded. Each major logical function in the kernel is manifest in an individual mechanism, and much effort was made to ensure a proper separation of concerns among these mechanisms.

2.5 IPC Functions in an RT-DOS Kernel

Real-time features of distributed operating systems are implemented at upper policy levels as server processes, and hence functionality of a DOS and RT-DOS kernel mechanism layer is almost similar. Therefore, existing DOS kernel implementations can be taken as example of both kinds. In most of the modern distributed operating systems the kernel essentially handles communication and some process management, and little else [TanenbaumA 1990]. The kernel takes care of sending messages, scheduling

processes, and some low level memory management. All of the remaining functions that are normally associated with a modern operating system environment are performed by servers, which are effectively ordinary user processes. That is why OS kernel is given names such as message passing kernel or communication kernel in these environments [Briat 1986]; and a message-based interprocess communication (IPC) mechanism is the essence of these kernels, at large.

At one extreme, the kernel could present the data-link layer of a local area network (LAN), letting programs tailor their protocols to their needs. Each process would specify destination machines and process names. At the other extreme, processes could transfer messages among themselves in a network-transparent and process-name transparent manner, providing a more abstract level programming interface. Accent [RashidA 1981], Demos/MP [Baskett 1977], and Charlotte [ArtsyA 1987], as well as the RT-DOS Kernel IPC, take this approach. From the communication mechanisms perspective, the majority of kernels in the front group would be described as "message passing" (MP) systems, while those in the second group would be thought of as having an underlying remote procedure call (RPC) model.

In all these kernels, connection-oriented reliable communication protocols, such as remote procedure call (RPC), reliable multicasting and broadcasting message passing services, are established on top of an unreliable connectionless IPC datagram service.

Because the major goals of the RT-DOS Kernel were flexibility and compactness, a great emphasis was placed on the design of mechanisms (as opposed to the specific policies concerning their use) in general, and on the design of the IPC mechanism in particular. Moreover, the topology of the RT-DOS system was designed specifically to provide necessary support for an efficient and high performance IPC mechanism. The RT-DOS Kernel IPC mechanism provides connectionless transport level communication protocols which were designed with group communication support such as multicasting [Mockapetris 1983], broadcasting [Kaashoek 1990], and unicasting in mind. It provides blocking type communication primitives which support remote procedure calls (RPC) [Hutchisson 1989, Tay 1990] between physically distributed processes, as well as non-blocking type of communication which is based on aforementioned group communication protocols. A more detailed elaboration of these issues is presented in **Chapter 5**.

3. THE RT-DOS KERNEL DESIGN CONSIDERATIONS

As the IPC implementation effort was a complementary part of the RT-DOS Kernel design project, this project history and its environment are highlighted briefly in this chapter.

3.1 General Objectives and Restrictions

The objectives of the RT-DOS project were: the specification, design and validation of a distributed operating system kernel which would support industrial applications [TayliA 1987]. The emphasis was on the identification of primary system functions and on the methodology of developing a layered reference model. The validation of the model and the kernel would be carried out by setting a testbed consisting of a series of microcomputers (Transputers) networked through high speed data communication channels.

The general context of the research was the area of distributed systems for real-time command and control applications. The aspects of these applications and the characteristics that set them apart from the other problem domains have been described in the previous chapter (**Chapter 2**).

At the completion of the project, it was expected to accomplish the following results :

- A layered reference model, for the specification of real-time oriented distributed operating systems;
- A complete implementation of a real-time distributed operating system kernel, conforming to the model specified above;
- Validation of the kernel with at least one real-time application.

It has to be noted here that the implementation of an IPC mechanism (which is the main subject of the thesis) for the above mentioned kernel is obviously one of the intermediate steps that should be taken at the initial stages of kernel implementation, as the rest of the other kernel components will be using it as a means of interaction between themselves.

In the context of the research, reliability of the kernel was emphasized much, in terms of providing predictable services (which rely on a collection of predictable IPC services) to upper levels, which is a basic requirement of a real-time command and control system.

Reliability can be defined as a degree to which the application goals continue to be met in case of failures, errors, and faults. Randell, at al. [Randell 1978] have surveyed the issue involved in achieving high reliability from complex computing systems and provided their definitions to help to distinguish between the reliability and the availability, and among failures, errors, and faults. To support the overall reliability goals of a distributed real-time command and control application, the system software must itself meet a certain level of reliability. In addition to this, the system must provide mechanisms that allow reliable applications to be constructed suitably. The kernel should not dictate a specific kind and degree of reliability, but rather it should allow its clients to choose what is desired for each individual set of circumstances at an appropriate cost [Randell 1978].

The distributed real-time command and control application domain calls for a set of kernel mechanisms that support the following reliability concepts:

- Correctness of actions (is a function of time, sequence and completeness);
- High availability of services (inversely related to the frequency of failures);
- Graceful degradation (is the property of a system that permits the system to continue providing the highest level of functionality possible, as the demand for resources exceeds its currently available capacity);
- Fault containment (is defined as a property that inhibits the propagation of errors among system components).

There are several issues which are not addressed by this research. Some of them are omitted in order to limit the scope of the initial effort to the aspects of this problem which are considered most interesting and important. Others are omitted in order to confine the target problem at a manageable level. The following are the specific issues that are currently not considered within the scope of the research :

- Heterogeneous microcomputer environment (interfaces of all Transputers with outside environments and each other are uniform);
- Communication sub network other than Inmos links [InmosA 1987];
- Specific performance goals;
- An existing operating system compatibility;
- Concurrent languages;
- Inter-network communication protocols;
- Security.

3.2 Transputer Hardware Architectures

The Transputer is a programmable VLSI device with communication links for point-to-point connection links to the other Transputers. It is a cross between a computer and a system building block, like a transistor, and it executes 10-30 mips [Transputer 1984, InmosA 1987, InmosB 1987, InmosC 1989, InmosD 1989, Datter 1985, InmosE 1987].

The Transputer family of 16-bit (IMS T212, T222) and 32-bit (IMS T414, T424, T425, T800, T805) processors boasts execution rates 2-10 times higher than those of standard microprocessors, while it occupies one tenth of their silicon area. The latest version of the family, T9000 Transputer [T9000 1992] which has some advanced features such as more than 200 Mbps link communication speed and 100 Mips CPU performance. It also incorporates 2-4 Kbytes of static RAM, a DMA interface, a timer, and built-in communication support (4 inter-Transputer links - INMOS Links), built-in micro coded kernel for multi-tasking and the construction of highly concurrent systems of many connected Transputers (**Figure 3.2(1)**). Lately, Transputers are used in building massively parallel super computers by connecting thousands of them to each other in different network topologies such as star, cube, ring, and combination of all. As an example, **Parsytec GmbH** company in Germany has manufactured the GC-5/16K supercomputers using more than 16,000 Inmos Transputers [Lamberts 1993].

Transputer Communications : The Inmos Links are the most obvious feature distinguishing the Transputer from other microcomputer-type devices. Each link provides two-way point-to-point connection with other Transputers. This allows arrays of Transputers to be assembled into multiprocessor systems without the problem of bus contention (**Figure 3.2(2)**).

Each Transputer link supports memory-to memory block transfer both for on- and off-chip memory. A link controller accepts a pointer and a block count when an OCCAM INPUT or OUTPUT statement refers to an inter-Transputer channel.

The DMA transfer is totally asynchronous, with message transfer taking place totally independently of the processor. Operating simultaneously, all the links can transfer data

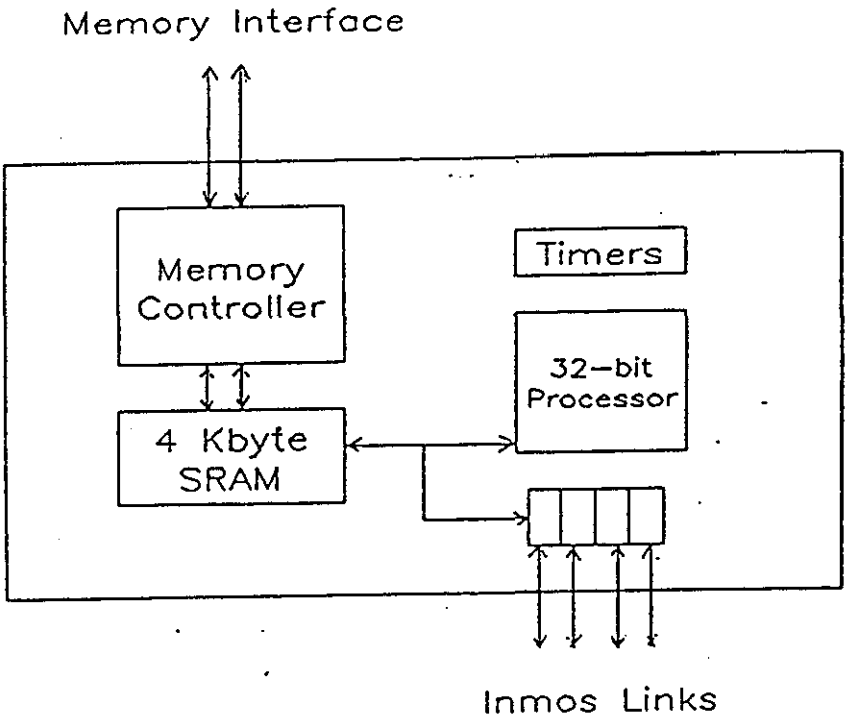


Figure 3.2(1) Transputer Block Diagram

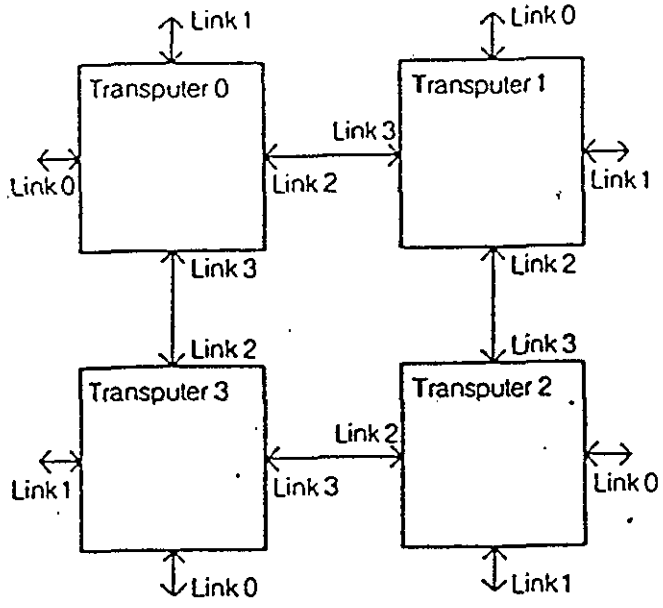


Figure 3.2(2) A Transputer Array Connected Through Inmos Links.

concurrently with processor execution, up to an overall throughput of 4x30 Mbits/s, depending on the selected speed of the links.

Regardless of the word length of the communicating devices, a message is transmitted as a sequence of bytes through the links (which are actually a pair of wires). For transfer in a single direction, the sending Transputer initiates traffic by transmitting a byte on one wire. The sender then waits for acknowledgment, which is sent through the other wire and which signifies that the receiving link can receive another byte and that process is waiting to receive it. The sending link reschedules the sending process only after it has received an acknowledgment for the final byte of the message.

For duplex communications on a single link, a Transputer interleaves the bytes it is sending with acknowledgment for the bytes it is receiving on the other link wire. An acknowledgment can be transmitted as soon as the reception of a data byte starts if there is a room to buffer another one. Transmission can therefore be continuous, without any delay between data bytes.

The links make Transputer systems as easy to engineer as possible. Regardless of internal performance, all Transputers use a 5-Mhz reference clock for approximate frequency information only - not for phase. This low frequency simplifies the clock pulse's distribution in large systems. All a system's Transputers do not have to be connected to the same clock, so Transputers can be connected in independently designed systems just as easily as TTL gates are.

An OCCAM channel provides a communication path between two processes [Tyrrell 1989]. Channels between processes executing on the same Transputer are implemented by single words in memory (internal channels); channels between processes executing on different Transputers are implemented by point-to-point links (external channels). For internal communications the compiler allocates the memory location (soft channels). For external channels' locations the bottom of the Transputer memory is used (16'80000000-16'8000001C where 16' indicates HEX). The implementation of external communications uses three separate registers to support autonomous DMA, leaving the processor free to work on another process. These link registers hold the count of the number of bytes to be transferred, a pointer to the location in memory (to input or output) and a pointer to the workspace of the process.

When control is transferred to the link registers the process is descheduled. Once the communication has been completed, the link interface signals the processor to add the process to the end of the list of active process. The *IN* and *OUT* instructions are used for communicating data; *IN* transfers a block of *A* bytes from channel *B* to the address pointed to by *C*, *OUT* transfers *A* bytes from the address pointed to by *C* to the channel *B* (**Figure 3.2(3)**). It is to be noted that *A*, *B* and *C* are Transputer operand registers. The processes at either end of a communication must have the same value in *A*, otherwise OCCAM compiler rejects the instructions treating such situations as errors.

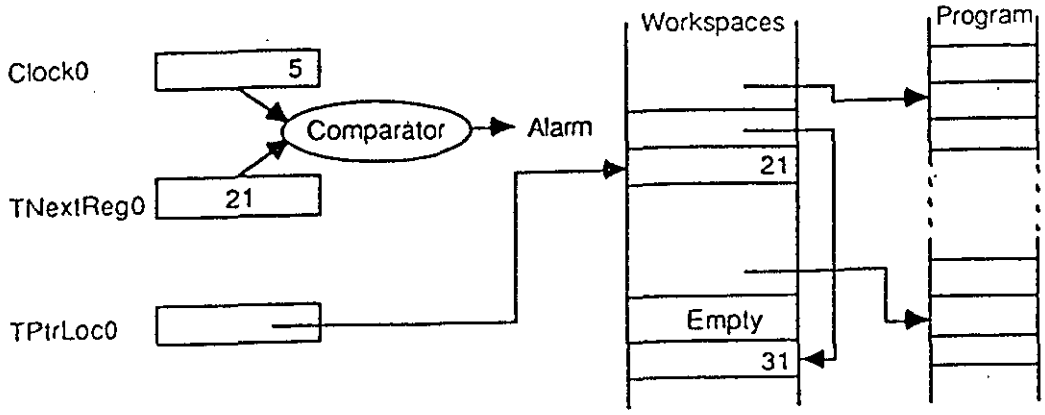
Memory Interface and Process Workspaces : The provision of 2-4 Kbytes of on-chip memory reflects a desire to use technology in a very optimal way. As transistors shrink in size, they can switch faster, so chips run more quickly. An on-chip RAM ensures that the Transputer spends most of its time executing at its rated speed from on-chip memory. When the processor does need to access off-chip memory, on- and off-chip access are identical as far as the programmer is concerned.

It is to be noted that some Transputers have built-in floating process units (FPU) and this adds extra complexity to the architecture. **Figure 3.2(4)** shows the internal datapaths of a IMS T800M Transputer.

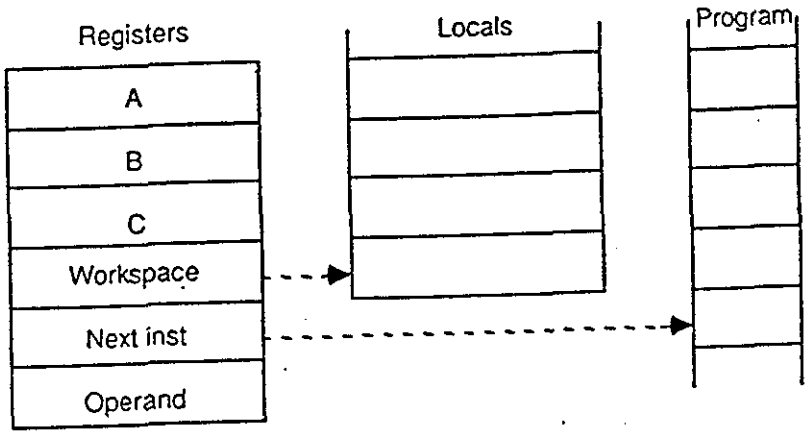
The address space is uniform, and all addresses are one word long, for a maximum memory of 2^{32} words on the 32-bit Transputers.

The memory interface has a world-wide multiplexed data and address bus. To control power dissipation- and because the 32-bit Transputer will in all likelihood normally be used in systems with large memory configurations- the bus outputs are intended only to drive external buffers, not a full complement of memory chips. For dynamic RAMs, address multiplexing is external to the buffers for different multiplexing requirements. The memory interface also supports both a simple cycle for ROM and RAM without address, and a multiplexed cycle for RAMs with multiplexed addresses.

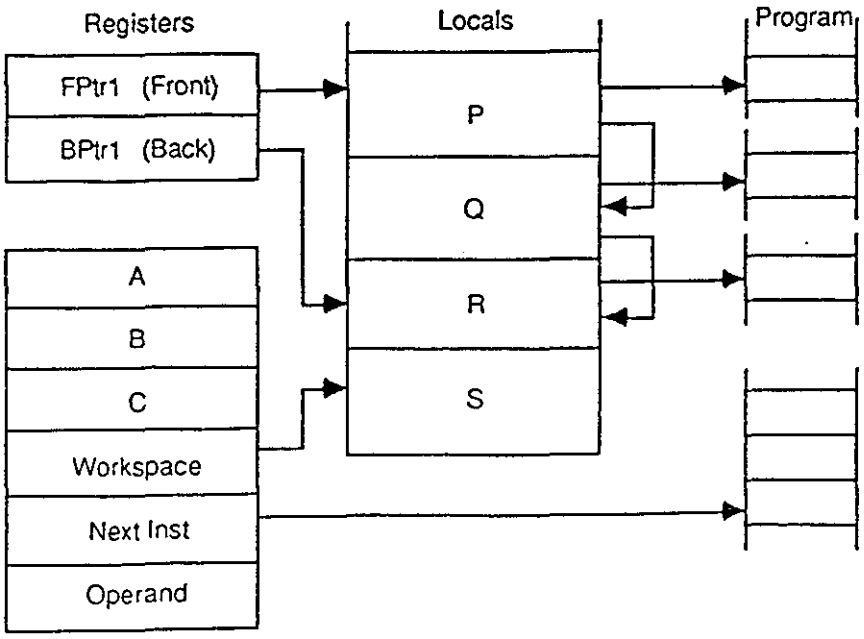
Each parallel process has a workspace associated to it , that is, a block of 32-bit words in memory (**Figure 3.2(3)**). The register W points to the beginning of this block during the execution of the process.



Timer Registers



Registers



Linked Process List

Figure 3.2(3) Transputer Processor Model and Instruction Registers.

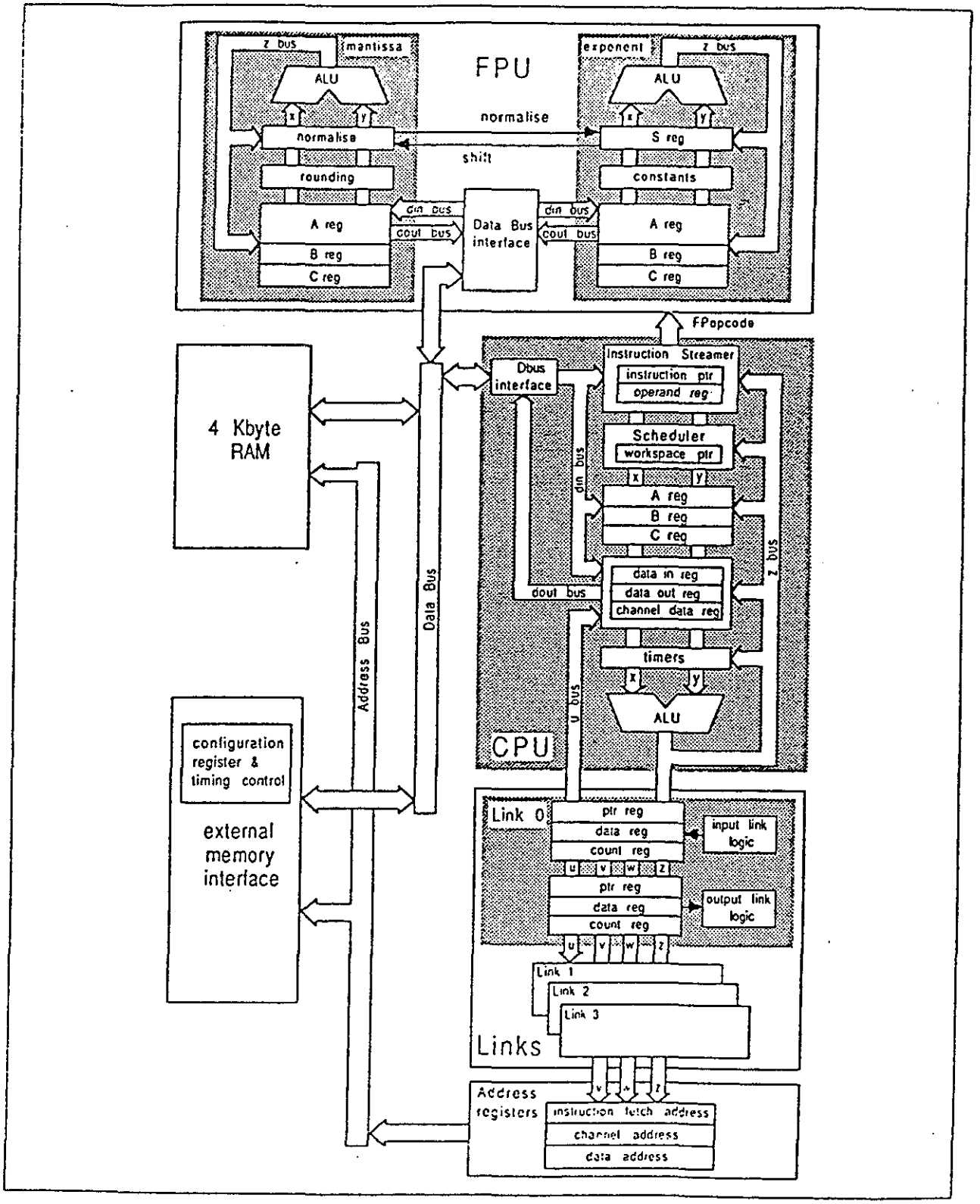


Figure 3.2(4) IMS T800M Internal Datapaths.

The workspace address and its priority level, given in the process descriptor, identify completely a process. The priorities of processes are stored in the least significant bit of the workspace addresses. Small negative offsets from the workspace pointer are used for storing information used on the process queues and information about communications and timers. Thus, when the workspace requirement is calculated, at compile time, room in addition to local variables has to allocated.

A real time kernel is hard-wired into the Transputer. A process is defined by its workspace address. Workspaces are linked to form two lists of waiting process (priority 0 = high and 1 = low). Special registers in the processor point to the front and back of the active process lists (**Figure 3.2(5)**).

A process is started by adding it to the end of the appropriate list. When the current process is descheduled it may have one of two states : (1) active or (2) in-active ((a) awaiting I/O, (b) time delay). In case (1) the process is placed at the back of the appropriate scheduling list. In case (2), the process will be added to the scheduling list only when the I/O is ready, or the time delay has expired. The process at the front of the list is scheduled. A high priority process will always be executed in preference to a low priority process. A low priority process will be preempted if a high priority process becomes available for execution. When there are no more high priority processes the low priority process will continue executing.

Seven locations near the bottom of the Transputer's memory map are used to hold the state of a preempted low priority process. One process, at most, can be preempted at one time requiring only seven 32-bit locations. The five main registers are saved with the status register of the preempted process and an internal register (Ereg) used in block moves. Workspace for parallel processes is allocated below the workspace of the parent. The first member of the PAR list (parallel processes of OCCAM) is allocated workspace immediately below the parent, the second immediately below it, etc.

Simple Instructions-Registers : One fundamental reason for the performance and implementation efficiency of the Transputer is its instruction set- which resembles that of a reduced instruction set computer (RISC). The Transputer has a compact program representation and micro coded instruction set. Hence, the Transputer does share RISC's advantages- an extremely simple instruction decode and few instructions [InmosE 1987].

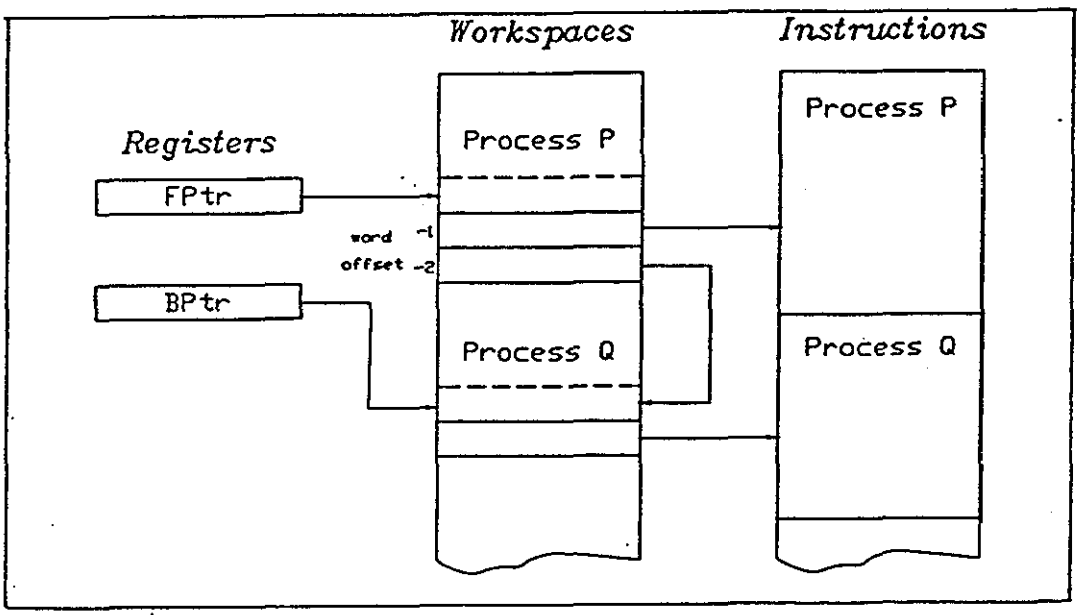


Figure 3.2(5) Typical Process Lists in Transputer Memory.

These instructions are independent of the processor's word length, which can be any number of bytes. The same instruction set is used for the 16- or 32-bit Transputers and could also be used for versions with other word lengths. Programs manipulating bytes, words, and truth values can be translated into an instruction sequence that behaves identically when executed by Transputers of different word lengths.

Each instruction comprises a 4-bit function code using the most significant bits and a 4-bit data value- for 16 functions and data values ranging from 0 to 15. Both one-address and zero-address instructions need to be able to use operands larger than the four data bits of an instruction. The two prefix instructions, one positive and one negative, extend the length of any instruction's operand. All instructions are executed when the four data bits are loaded into the four least significant bits of the operand register which is then used as the instruction's operand. All instructions, except for those of the prefix, terminate by clearing the operand register in preparation for the next instruction.

The prefix instruction loads its four data bits into the operand register and then shifts it up to four places. The negative prefix instruction is similar but complements the operand register before shifting. Consequently, operands from -256 to +255 can be represented with one prefix instruction, and a sequence of such instructions can extend an operand to arbitrary widths- constrained only by the operand register's width. These simple prefix instructions have profound consequences for performance and compatibility. First, they are decoded and executed just like any other instruction, and that procedure simplifies and speeds instruction decoding. Second, they simplify language compilation by providing a completely uniform way of allowing any instruction to take any size of operand, up to word length of the processor. Finally, they allow operands to be presented in a form independent of the processor's word length.

As it can be seen in **Figure 3.2(3)**, a 32-bit workspace pointer *W* points anywhere in memory, making context switching easy and fast (0.002 ms). Three 32-bit registers *A*, *B*, *C* form an evaluation stack and should not be considered as a set of independent registers as on many other microprocessors. Registers are defined by the user; the compiler allocates room inside a process workspace. Parameters, even temporarily, are never kept in the evaluation stack. Instructions which do not use *A*, *B*, or *C*, like *JUMP* instructions, leave these registers in some unpredictable state. The usual condition code register

does not exist : the results of a comparison is left in register *A*. An error flag *E* and a *'halt on error'* flag *H* exist for handling overflows.

The processor maintains two scheduling linked lists for simulating parallel operations. There are four registers specifically for forming these linked lists; two registers for high priority processes, one pointing to the front of the list (*FPtrReg0*), the other to the back (*BPtrReg0*) and two registers for the low priority process (*FPtrReg1*, *BPtrReg1*). There are six registers and two bits used for the local operations on the Transputer : Two clock registers, *ClockReg0* and *ClockReg1* (one for each priority level), two registers pointing to the first items on the two priority timer queues. *TPtrLoc0* and *TPtrLoc1*, and two registers indicating the time of the first event to occur, *TNextReg0* and *TNextReg1* (again one for each priority level). The two bits indicate whether there is anything on either of the timer queues, *TEnabled0* and *TEnabled1*.

3.3 Transputer Software Development Tools

In this section, major software development tools for Transputer based systems in general, and the ones which have been used in the RT-DOS Kernel design in particular, are explained briefly. As it can be appreciated from the following arguments, capabilities and limitations of the chosen development tools are becoming major factors in the success of the design and implementation strategy of the target system.

INMOS TDS D701 Transputer Development System : The INMOS TDS D701 Transputer development system [InmosH 1990] consists of an IBM PC add-in board (IMS B004) and the related software. The board enables users to evaluate and demonstrate the use of Transputers. Containing a 32-bit Transputer (T414) and 2 MB on-board RAM, the board provides a powerful upgrade to the IBM PC XT or AT.

The TDS software actually runs on the Transputer (in collaboration with a small program executing on the IBM PC which provides access to the PC's resources) in the 2 MBytes of RAM, offering an extremely fast and efficient compilation.

The PC I/O channel is interfaced to an INMOS link by on-board IMS C002 link adapter, and logic is provided for the Transputer's external reset control and monitoring. Simple external connections complete the communication and control route between the IBM PC and the Transputer system.

Though the IMS B004 board is supporting a number of different software development systems, TDS D701 is based on OCCAM language and related software, including an folding editor, compiler, linker, network loader and configurer, and some debugging utilities. OCCAM language's capabilities and limitations are explained below in detail.

The OCCAM Language : OCCAM is a simple but yet powerful programming language which enables a Transputer system to be described as a collection of processes which operate concurrently and communicate via named channels [Geraint 1987, InmosF 1988, May 1986]. It is designed to support concurrent applications in which many parts of a system operate separately and interact.

The novelty of OCCAM is in its treatment of concurrency. OCCAM enables the programmer to express a program in terms of concurrent processes which communicate by sending messages through communication channels [Hull 1989]. This has two important consequences. Firstly, it gives the program a clear and simple structure as the individual processes operate largely independently. Secondly, it allows the program to exploit the performance of many computing components, as each concurrent process may be executed by an individual processor. OCCAM can capture the hierarchical structure of a system by allowing an interconnected set of processes to be regarded from the outside as a single process. At any level of detail, the programmer is only concerned with a small and manageable set of processes.

An OCCAM program is constructed from processes combined together using keywords called process constructors. The internal structure of each constructed process is indicated by a fixed layout with each component process appearing on a new line, indented from the keyword that introduced the whole construction. The primitive processes (which are SKIP, STOP, assignment, Input, and Output) form the level of processes in an OCCAM program [Geraint 1987, Pountain 1987].

An OCCAM program can describe several processes to be run concurrently and the processes may communicate by passing messages. OCCAM processes do not use shared variables, nor semaphores, "critical regions" or "mutual exclusion zones". OCCAM does not require (nor support) shared memory. Data can be exchanged between processes only by passing messages through the OCCAM channels. There are no

multiple senders or receivers, no broadcasting, and no uncertainty about where a message come from or where it is going. Messages are unbuffered, so sending and receiving a message involves momentary synchronization between the two participating processes. Messages are sent through static channels, as if through a circuit switched (rather than packed switched) network [Wexler 1989].

The concurrent processes that form an OCCAM program communicate via channels, using the primitive processes of *Input* and *Output*. The *Output* process

Channel ! expression

sends the value of the expression over the *Channel*. Similarly, the *Input* process

Channel ? variable

receives a value from the channel and stores it in the variable. The communication is synchronized, and only takes place when both processes are ready.

The OCCAM Input and Output processes have an exact parallel in the Input and Output statements of CSP [Hoare 1978], even to the extent of using the same symbols "?" and "!". The parallel construct

```
PAR
  Out1 ! x
  Out2 ! y*y
```

will output the value of x over channel *Out1*, and will output the value of $y*y$ over channel *Out2*. The program will simply execute the first process that is ready. For a parallel composite process to be legal, none of its component processes may change the contents of any variable that is used in any other component as there is no shared memory in OCCAM programs.

In the following OCCAM alternative constructor

```
ALT
  In1 ? x
    Out1 ! x*x
```

```
        In2 ? y
          SKIP
```

if x is ready for input from channel *In1* before y is ready for input from channel *In2*, then $x*x$ is output on channel *Out1*; otherwise SKIP is executed (i.e., nothing is done). In OCCAM, loops are formed using the repetition constructor.

The following constructor (loop control structure)

```
    WHILE x > 0
      SEQ
        In1  ? x
        Out1 ? x
```

will execute an OCCAM sequential construct SEQ (input x from channel *In1*, and then output x to channel *Out1*) as long as x contains a positive value. A process can be made to execute continuously by the constructor *WHILE TRUE*. The basic concurrency requirements of communication, synchronization and process creation are built into OCCAM at a high level, rather than being implemented through low-level error prone devices such as semaphores. It is this factor which makes OCCAM so useful in describing concurrent systems and algorithms.

Though Transputer assembly language code can be embedded in OCCAM programs (and OCCAM can be a harnesser for other languages, such as C), in that case the reliability and robustness of OCCAM programming model is lost. Programming under this model requires that the source code is specifically tailored to, and the executable code configured for, a fixed hardware configuration [Oakley 1989]. Applications can not be configured to the resources available at run time. The model offered by the Transputer and OCCAM has some other disadvantages, namely deadlock and livelock, which are not considered to be any concern of the operating system or its built-in micro kernel.

Because of these limitations current implementations of OCCAM and its development environment is not convenient for operating system implementation.

INMOS ANSI C Toolset and Other Language Compilers : The current implementation of OCCAM and its Configuration language extensions impose a relatively rigid process structure at the source code level [Oakley 1989]. Because of these and other limitations of the OCCAM development environment, the OCCAM based software development tools were replaced by the INMOS ANSI C based software toolset [InmosG 1990]. Parallel C offered by 3L and other C compilers (INMOS ANSI C) offer the ability to 'flood fill' a Transputer array with identical worker processes; and runtime processes establish what hardware resources are available, automatically place copies of the worker code on each processor, then handle message passing between the workers and their master process.

The Inmos ANSI C toolset has been designed to reflect the processing model of communicating sequential processes (CSP). The inherent flexibility of the C language, the capacity to mix code from different languages, and the ability to use the concurrency features of the Transputer make it a powerful tool for programming concurrent systems. Considering OCCAM language limitations, the Inmos ANSI C has been chosen as implementation tool because of its flexibility for the RT-DOS kernel project. The Inmos ANSI C compiler and its supporting tools run under DOS (there are versions for other operating systems as well, such as VAX/VMS, Sun OS, and PC-DOS), either on the host itself or on a Transputer board attached to the host; and can be used in conjunction with existing text editing software and source control systems. For this reason, no editor is provided with the toolset, in contrast to the OCCAM TDS Toolset which has a sophisticated one.

The toolset has a ANSI C compiler (***icc***) with concurrency support which generates object code for specific Transputer targets, a configurer (***icconf***) to analyze the configuration description and produce configuration data file for code collector, a code collector (***icollect***) to collect linked units into a single file for loading on a Transputer network (takes as input a configuration data file or a single linked unit), a file format converter (***icvlink***), a network debugger (***idebug***) to provide post-mortem and interactive debugging of Transputer programs, a memory dumper (***idump***) to debug programs that run on the root Transputer, a memory configuration tool (***iemit***) to evaluate and define memory configurations for later incorporation into ROM programs, an EPROM program formatter tool (***ieprom***) to format Transputer bootable code for input to ROM programmers, a toolset librarian (***ilibr***) to build libraries of compiled code in the same

format as the C runtime library, a toolset linker (***ilink***) to resolve external references and link separately compiled code into a single file, a binary lister (***ilist***) to disassemble and decode object code and display information in a readable form, a makefile generator (***imakef***) to generate makefiles for input to MAKE programs, a host file server (***iserver***) to load programs onto Transputer hardware and provide runtime access to the host, a T425 simulator (***isim***) to simulate program execution on an IMS T425 Transputer and provide simple debugging facilities, and finally a skip loader tool (***iskip***) used with ***iserver*** to load programs onto external networks over the root Transputer.

The ANSI C toolset can be used to write programs targeted at IMS M212, T212, T222, T225, T400, T414, T425, T800, T801, and T805 Transputers. Code can also be written to run on a group of processor types by compiling for a Transputer class. Transputer assembly language code can be embedded in C programs when required. A full range of high level language constructs including replicative and conditional statements make it easy to explore different configurations before committing to hardware.

3.4 RT-DOS Kernel Design Approaches (Object Model and Layered Model)

Design methodology for distributed systems in general, and real-time distributed operating systems in particular, have not matured yet [StankovicD 1985, Turek 1992]. The RT-DOS Kernel research project [TayliB 1990] team initially faced the critical decision of adopting a suitable methodology for the design and implementation of the kernel. The object-oriented paradigm was investigated as a potential alternative to the hierarchical layered system model [BorA 1989, BorB 1990]. Though the object oriented methodology is generally used in the implementation of databases, programming languages, and in complex systems software analysis and design; the main point of interest in the RT-DOS project was exploring its usage in the design of the Real-Time Distributed Operating System (**RT-DOS**) kernel which runs on a Transputer-based network.

During the preliminary design of the research project "Real-Time Distributed Operating System Kernel" (**RT-DOS**), the investigators felt the necessity to explore the object oriented model as an alternative design and implementation strategy to the layered model. The study focused especially on the application of the object oriented approach to

the operating system field. Other application domains like databases, programming platforms are voluntarily omitted to keep the investigation within the manageable limits.

A detailed and extensive report has been presented on the issue, as a compilation of the surveyed literature [BorA 1989]. It presents the object oriented paradigm, its chronological evolution, and introduces basic abstractions and concepts of the model. Pros and cons of the object oriented approach are also discussed as a programming methodology in general, and as a design alternative for the RT-DOS kernel implementation in particular. A number of recent implementations that used object oriented approach as design methodology or provided their users with object programming interface have been investigated in detail. Particularly, the Alpha kernel [Nortcutt 1987] proved to be a convincing and impressive argument for the object oriented approach. Besides its implementation methodology, the Alpha kernel also presented innovative design features as a real-time distributed operating system kernel. Therefore, a substantial part of the document was devoted to the Alpha kernel and its implementation details.

The following three sections are devoted to the elaboration of the issue, comparison of the object oriented model versus classical layered approach; and at the end, the reasons why the object oriented paradigm was not chosen as the implementation methodology, is presented.

3.4.1 Object Model

"There is little hope of mastering the complexities of modern graphical environments without the leverage that object-oriented programming provides" [Davson 1989]. "Object-oriented technology is on the march and will doubtless soon appear in other forms" [Udell 1989]. Expressions similar to these quoted sentences figure nowadays in scores of articles. Though the object model became popular only in the last decade, accepted as a programming methodology after the implementation of the Smalltalk language, it has effectively been used as an extension for existing programming languages in complex software systems development, databases, and in operating system implementations since 1950s. Recently, an ever increasing number of real-time and distributed operating systems, practicing the object-oriented paradigm, have started to emerge from research laboratories.

It has been argued that there are two important reasons for the object model's growing popularity, especially with distributed operating system designers. First, there are many issues involved in operating systems design all closely interrelated e.g. naming, protection, atomicity, resource management, synchronization etc., and the object model seems to take account of their interdependencies, encapsulating many of the problems in a single abstraction. The object model is also claimed to handle networking gracefully, supporting the implementation of reliable mechanisms given that techniques for performing remote operations on objects are well understood and can be made very efficient.

It is not easy to understand and use the main ideas hidden behind basic abstractions of the object oriented model without using a real object-oriented language, such as Smalltalk or C++. In addition to this, it appears that the reasons for which these abstractions were developed and used are embedded in the chronological progress of the method itself. Therefore, a rather wide range of the implementation samples was examined to understand the capabilities and limitations of the model and assess its advantages and disadvantages.

What is an Object Oriented Model: The object-oriented model has been introduced by a number of authors [Dyke 1989, BYTE 1986, Danfort 1988] as :

- A vision, or a way of organizing a system description. The central intuition in the object-oriented vision is that systems can be built by describing sets of related objects and that objects have attributes and behavior;
- A set of programming techniques that specifically include facilities to manipulate objects, attributes, and behaviors;
- A large complex system that enables and encourages using object-oriented programming techniques.

In the programming languages context, the term "object-based" is used for languages supporting only objects; and "object-oriented" for languages which are supporting objects, classes, and inheritance.

This section covers the object model in its generality, and discusses the requirements for a software system to be classified as object-oriented or object-based.

Historical Background of the OOP: Part of the object oriented terminology such as objects, messages, classes come from ALGOL and SIMULA languages, but object-oriented programming can trace its roots to the earliest uses of the computer, yet it is thought of by many as a new programming methodology. The object model has been implemented as a programming language (Smalltalk [Goldberg 1984], Strand88), and its ideas and constructs have been added to existing languages (Pascal, C++, Ada, Modula). Object-oriented programming is a methodology that employs data abstractions (a data abstraction is a way of defining data considering the way it may be used, rather than what it is), called objects, as the basic structure of programs [Dyke 1989].

In the USA, the Minuteman missile project was designed in 1957, using primitive object-oriented structuring techniques. The design was divided into a handful of related discrete components, every component dealt with one aspect of the design and was created by a specialist who had a unique expertise, for no single individual had the breadth of knowledge to create the whole program. Each of the program components was encapsulated, that is, it had its private data and was virtually a separate program with its own methods and procedures that would apply to those data. By sending data and commands between the components and by using key design parameters supplied by the operator, the computer program was designed and simulated the flight of experimental missiles in the computer. Similarly, in object-oriented programming, the objects are made up of private data and methods, and they communicate with each other through the use of messages which may contain data arguments. In this way, object-oriented programming is like having a group of specialists working together to solve a problem. A system, whose functions are formalized and generalized so as to apply to many different kinds of objects, keeps track of the status of each of the objects and determines which information to send and where [Dyke 1989].

In 1960s, SIMULA67, an Algol-based language introduced the concept of class, which is the implementation of a data abstraction through encapsulation, and the concept of a class hierarchy to permit the inheritance of methods. Using SIMULA67 as a spring board, the language Smalltalk was developed at the Xerox corporation's Palo Alto Research Center. Smalltalk has further formalized the notion of objects and message passing among objects. Since then, Smalltalk has been considered to truly be a real object-oriented language and the first example of this type that has all the important

features of the object-model. Smalltalk offers the generality and conceptual wholeness of a pure object-oriented language, in which everything is either an object or a message [Udell 1989].

The C++ [Stroustrup 1986], EIFFEL, CTT, Mentat [Grimshaw 1993], and Objective-C languages are object-oriented extensions of the C language. FLAVORS is an object-oriented extension to LISP, and LOOPS and KEE are programming environments that use object-oriented programming as functional building block. Strand88 is a truly object oriented language designed for the development of parallel programs, running on a wide range of hardware including Transputers. CHAOS [Bihari 1992], an object-based language and programming/execution paradigm, has been designed for dynamic real-time applications which require complex embedded systems.

It is also possible to consider object-oriented programming as a style, rather than a language; and, the object-oriented programming can be built using existing procedural or functional languages.

Object-oriented programming concepts have also been used as the organizing principle for application development, and to supplement existing languages for application development and debugging (for example Turbo Pascal 6).

Object-oriented programming concepts may be implemented in varying degrees of completeness. Numerous implementation examples (mostly on different aspects of operating systems design) which use object-orientation as a programming paradigm, are briefly presented below.

Though the object oriented model became popular as a programming language concept, it has been used in the design of database software, because it lends itself to the implementation of an atomic transaction concept, a desired feature for implementing such issues as atomicity to provide data consistency and reliability, especially in distributed database environments. To restrict the main domain of interest to a manageable size, the object-oriented database systems implementation examples were excluded from the survey. Implementation of atomic transactions, which are supporting reliability and fault-containment, is discussed in relation to the Alpha kernel [Nortcutt 1987] in detail, as this issue is one of the reasons for the adoption of the object-model.

Basic Abstractions and Mechanisms of the OOP Model: Recently, a number of researchers discussed the requirements of an analysis process for object-oriented software as an alternative methodology to the standard structured analysis approach [Balin 1989]. Before trying to give a solid picture of the object model, it is useful to summarize these discussions comparing the object model approach against the traditional methods of system analysis and design. Parties involved in these discussions claim that object oriented programming methodologies are mainly based on decomposing processes into objects (active and passive) and allocating them functions until :

- Every known functional requirement is met by one of the objects;
- The internal state of the system is adequately represented by the states of all objects.

Structured analysis is a method of articulating functional requirements. Any system must accept certain inputs and deliver certain outputs. Processes are introduced to represent transformation of inputs to outputs. The main principle of aggregation in this method is that functions are grouped together in a process if they are constituent steps in the execution of a higher level function. The constituent steps may operate on entirely different data abstractions.

In object-oriented design the main principle of aggregation refers, to the underlying data. Functions are grouped together if they operate on the same data abstraction. Namely, functions executed in sequence can reside in different objects which include relevant data. In concurrent execution environments performance suffers mostly because of common data access rather than functions. Some authors also claim that grouping functions using common data in one object will reduce the cost of data/control flow and increase the concurrency in the overall system [Nortcutt 1987].

Principles of data encapsulation and information hiding are the main requirements for a method to be object-oriented. Information hiding and data abstraction can be summarized as "*give only as much information as it needs to know in order to carry out its function correctly, and hide as much information as possible away from the module*". In this way, information passing between modules and the complexity will be reduced. For a language, encapsulation means that the language provides a way to combine data and the code that operates on that data into reusable structures.

In the object-oriented method the emphasis is not on the transformation of inputs to outputs but on the underlying content of the entity. Entity content can be considered as:

- Data structures that define the entity;
- Underlying state of a process as it evolves in time (data stack);
- Aspect of a process that is persistent across repeated execution cycles (invariants).

An object-oriented system can be thought of consisting of an upper hierarchy of entities, decomposing at lower levels into functions. To complement the notion of entity a function is purely a transformation of inputs to outputs. It has no underlying state that it remembers across successive invocations. Every function must occur in the context of an entity, it must be performed by or act on the entity. An active entity is one that operates on inputs to produce outputs. A passive entity is one that is acted upon. Passive entities may consist of more than just data; they may provide a set of primitive functions for accessing the data. Abstract data types such as lists and stacks are the best-known examples.

Though the terms about the basic abstractions of object programming are not commonly accepted, the kinds and basic functions of these abstractions are almost the same in all implementations. First abstraction is passive entity (object in Smalltalk, module in Module-2, package in Ada, or entity) which includes encapsulated data, and access and synchronization routines to this data (functions, methods, operations in Alpha, etc.). Second abstraction is active entity which carries control and status information, and represents the computation in the system (thread in Alpha, process, procedure, etc.). The last abstraction is a communication mechanism (message passing system) between the active and passive entities (operation invocation in Alpha, method calling, function calling, message passing, etc.).

As an introduction to the object model, a comparative look at the *Process* abstraction in the conventional process model versus the *Thread/Object* abstraction of Alpha, is presented below (please note that the term "object" as in object-oriented, will be defined later on). **Figure 3.4.1(1)** depicts corresponding parts of a process and its Alpha counterpart. In Alpha, the design of threads and objects closely mirrors the kernel's programming abstractions by splitting typical processes into two independent

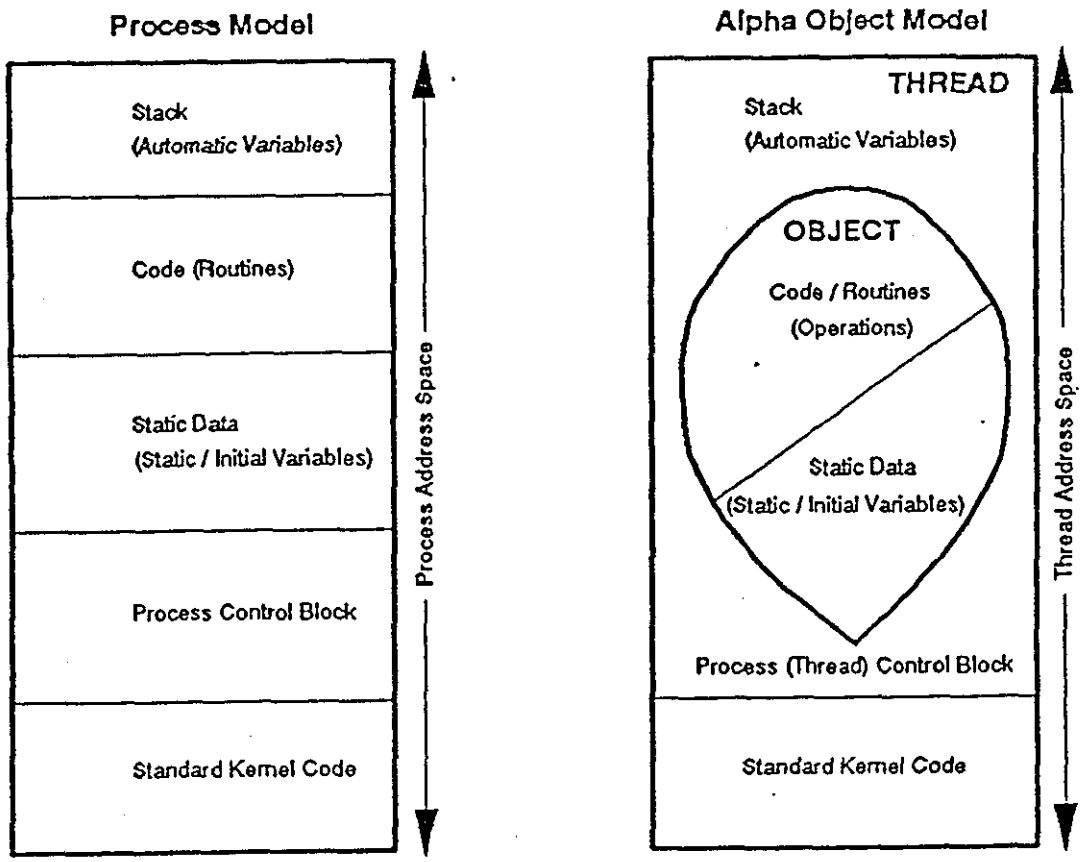


Fig 3.4.1 (1) : Comparative view of Process Model versus Alpha Object Model

components : a passive object part and an active thread part. Objects are quite similar to the code and data portions of traditional processes, that is, process without a stack segment or process control block. Objects consist of regions of data that the object serves to encapsulate, regions of code that are used to perform the operations defined on the object, and the various control structures used by the kernel to manage the object. The data portion of objects consists of three subparts :

- Statically allocated, uninitialized data;
- Statically allocated, initialized data; and
- Dynamically allocated, uninitialized data (i.e., heap storage).

The data part of an object represents only the global data associated with the object; all local data (e.g., automatic variables of subroutines) are provided on a per-thread basis and are not associated with the object proper. The code for standard operations is part of the kernel and is shared by all objects. Objects are passive entities and there is no activity in an object until an operation has been invoked on it.

In this model, threads provide the components of a typical process other than those provided by objects, such as code and persistent data. This consists primarily of execution stacks and system control information. The major component of any thread is the portion that contains the client's stack. Threads provide each operation invocation with a separate stack (to support object migration) that may be used to store any automatic variables declared within the scope of an operation. This stack space is reclaimed when the invoked operation completes. Each stack is protected so that a thread can only access the variables associated with the particular operation execution under way at any point in time. A thread has another part that contains the thread's kernel stack, and a part that contains the thread's invocation parameter pages.

Objects: An object is an entity with a private memory and a public interface. Messages are used to instruct an object to report on or alter its private memory. Messages are implemented by procedures (i.e., methods, functions, operations) that have special privileges in accessing the object's private memory. An object consists of both private data and the methods that can act on that data.

Each object is a triple : Unique-name, Type, Representation [Polymorphism 1985]. The unique-name distinguishes an object from all other objects. The type of an object defines the properties of the object (e.g., program, stack, process, etc.). The representation of an object is hidden from the users view within a type manager which implements the operations of the type definition [Goldberg 1984]. **Figure 3.4.1 (2)** illustrates an object structure consisting of data, code (standard operations that can be applied on object), and entry points.

There are two schools of thought and practice [Balin 1989] concerning the definition of complex objects :

- Abstract data type based definition; and
- Inheritance-based definition (simple inheritance in Smalltalk, multiple inheritance in KEE, etc.).

Abstract Data Type-Based Object-Structuring : An object-oriented software system is an assembly of objects. Each object will satisfy the Parnas module criteria. That is, a module (e.g. the type manager of an object) implements an abstract data type and performs all of the required actions on its data, and specifies the necessary pre-and post-conditions for acceptance of the results of those actions. The functions of a module are made available to other modules as procedure calls, and these procedure calls constitute the only access to the functions of the module. In particular, the data manipulated by the module is only made available to other modules by procedure invocations; other modules have no direct access to the location or the representation of any data used by the module. Modules detect conditions which violate their specifications and prevent application of functions upon the module's data when the necessary conditions are not met [Stroustrup 1986].

Inheritance-Based Object-oriented Structuring : In an inheritance-based object-oriented system, structures and types are defined as extensions of existing types [Mayer 1988]. A type definition begins with functions inherited from parent types. Complex types are defined through multiple inheritance. Inheritance is to receive properties or characteristics of another, normally as a result of some special relationship between the giver and the receiver. Frequently this is referred to as a class hierarchy.

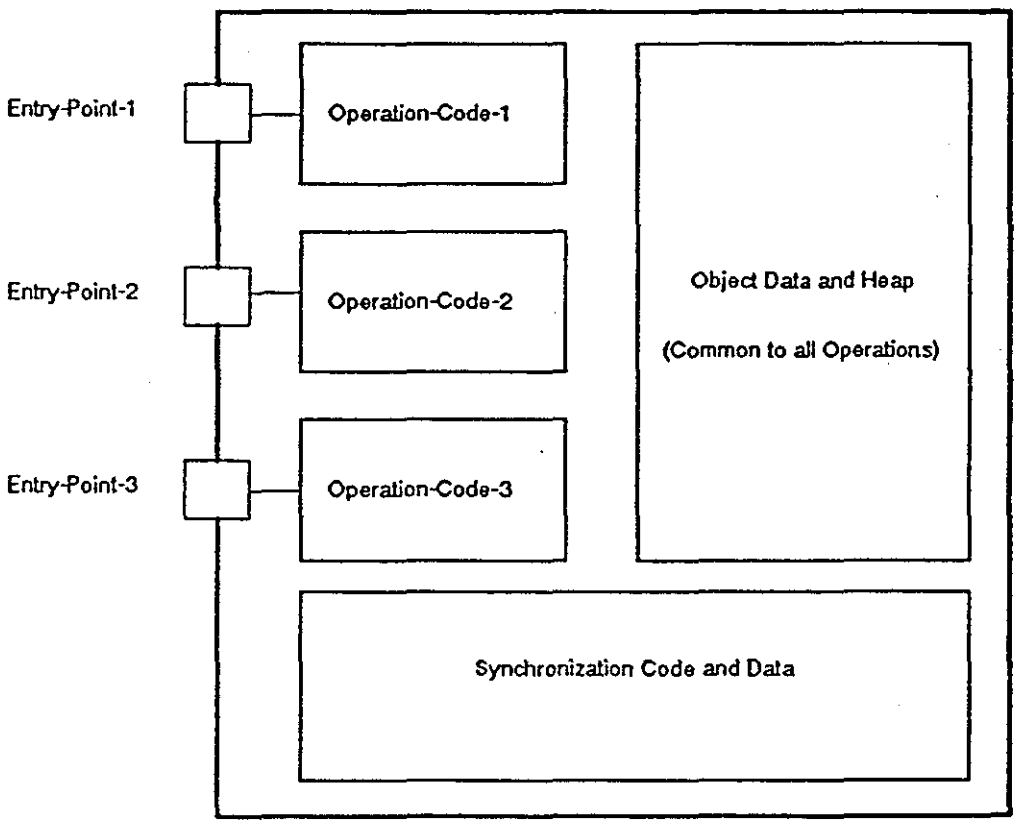


Figure 3.4.1 (2) : Object structure

When using inheritance in this way, an object is specified as what is new about the new object in reference to a class or subclass object definition. In Smalltalk, a subclass inherits and may extend both the state representation and the operations of the superclass [Stroustrup 1986].

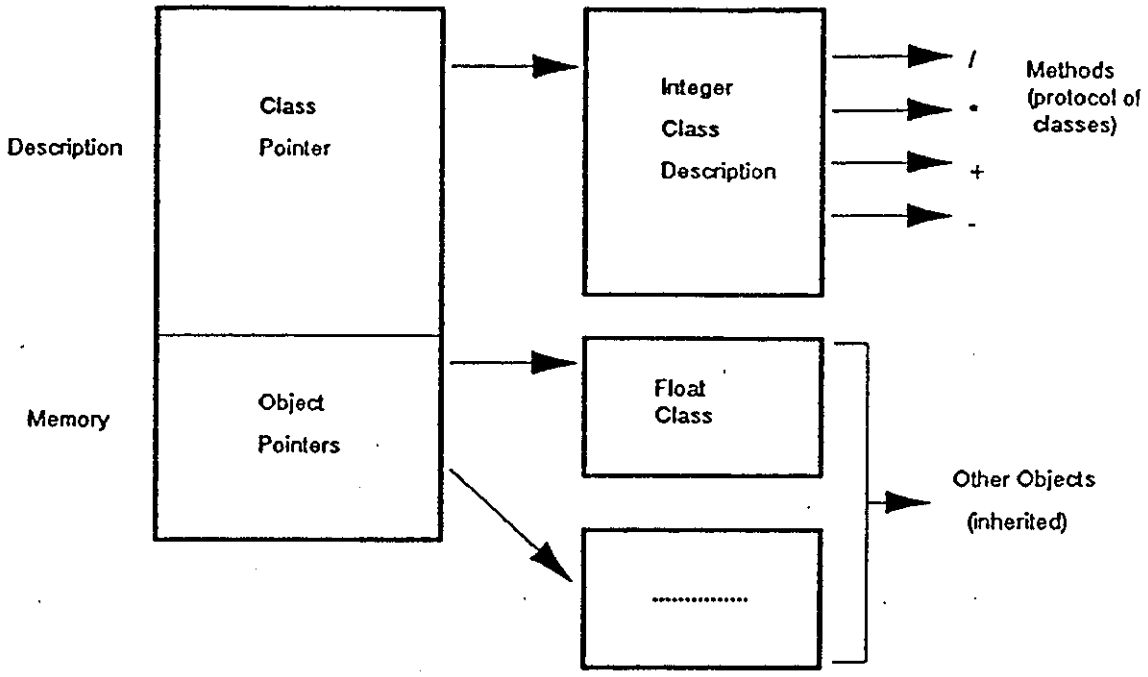
A protocol (method in Modula, function in C++, operation in Alpha, etc.) is provided as part of every class definition. Any method associated with a super class is inherited by a sub class. For example, numbers respond to a variety of arithmetic messages inherited from class *NUMBER*. In **Figure 3.4.1(3)** the inheritance feature of objects in Smalltalk is shown.

Abstract Data-Type, Inheritance and *Vertical Partitioning* : Inheritance is not compatible with vertical partitioning for several reasons. It tends to inhibit definition of type specific semantics for state management functions and blur the concept of strong atomicity at manager boundaries as required by vertical partitioning. Abstract data type definitions of objects, on the other hand, provide a natural framework for implementation of both of these requirements for vertical partitioning.

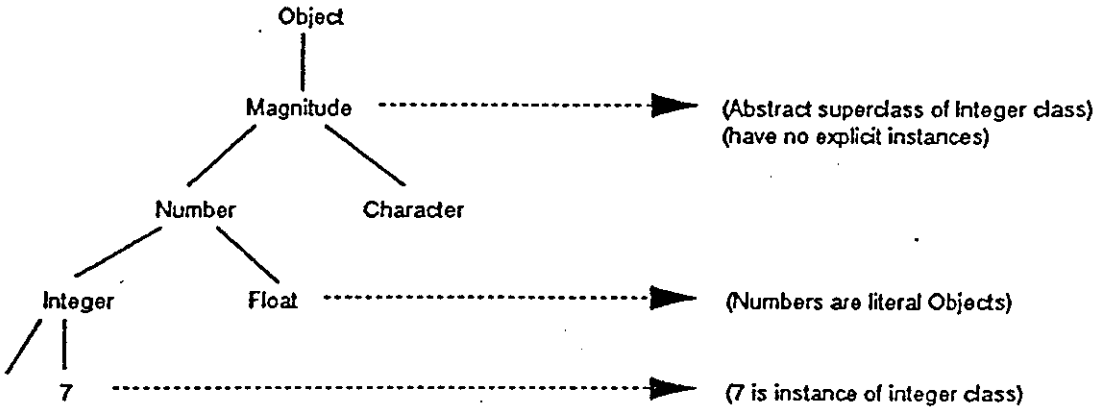
In an object-oriented database, programs (methods / operations) are viewed as objects and thus can be moved around the distributed database just like any other object. When performing a computation, the system can move data to the program or vice versa.

Objects may be active as in the Actors paradigm, or passive as in CLU, or somewhere in between as in Simula or Smalltalk in terms of initiating actions [Wiederhold 1986]. Compound objects can be thought of as lightweight tasks that share the same address space.

In this thesis, it is assumed that an object is any passive element of a computation whose state can only be modified by members of a well-defined set of operations or functions. The behavior of an object is completely determined by its set of permissible operations. The example of an elevator can be given as an abstract object on which only three operations are defined : install, which initializes its state, plus up and down, used by passengers to change floors. In a programming methodology, objects with the same set of defined operations are said to constitute a type.



b) A typical Integer class object structure



a) Superclasses and Subclasses hierarchy

Figure 3.4.1 (3) : Inheritance feature of objects in Smalltalk

The appropriate granularity of objects is related to the cost of inter-object communication. The cost of inter-processor communication in a distributed computer system suggests the use of medium-to-large-scale objects. In Alpha, an object is 100 to 10,000 lines of code in C. The object model of programming suggests that large amounts of data should not be passed as parameters to invocations of operations on objects. It is to be noted that, message passing between objects in Alpha and other systems is the same as procedure call in Pascal [Shipman 1987].

In the operating system context, named objects are generally associated with memory segments, and access is controlled by means of a capability. Objects are referenced by unique system-wide names. Encapsulation provided by the object abstraction appears to be a promising approach to the design of fault-tolerant and protected operating systems [Kohler 1981], to support the features such as fault-containment, atomicity, gracefully degradation, availability of services, and consistent behavior of actions.

Operation Invocation on Objects: In the object model all computation is performed by invoking functions on objects. A form of abstraction, called aggregation in the database literature, provides a natural means of building high-level abstract objects and functions from lower level primitives. This relates to the layered system approach advocated by operating system designers, where every layer interface can be represented by a collection of abstract objects and possible operations on them. Information on the internal structure of the objects and operations is inaccessible to their users [Kohler 1981].

An object consists of data that are tightly coupled with all of the operations that can act against it. Those operations are often referred to as methods (operations in Alpha), and the communication between objects requesting that some action take place is often referred to as sending a message (operation invocation, object invocation, or applying methods).

Object Types - Classes and Inheritance: Inheritance is the ability to derive a specialized structure from a more general one (objects inherit data and methods, add new ones, override old ones if necessary). All objects belong to a class (i.e., a type) that defines the messages that the object can understand and respond to. A class inherits all the messages from its superclass(es).

A class is a template from which objects are created (as a result of object instantiation, replication, and activation). Inheritance is a property of classes that allows them to share resources. Classes may be arranged in a hierarchy from most general to most specific. Classes lower in the hierarchy may inherit methods and attributes from classes above.

In object-oriented programming, a piece of data should not have its type or meaning determined by its location in a list. This principle distinguishes objects from data (encapsulation). At some future time, a programmer may change the internal representation of data in a class. If, as a result, any attribute of this class became the second or fourth item, requesting the third to get that information would yield an incorrect result. Users of data would then become responsible for understanding changes that take place in internal data format. All methods using the message to obtain any attribute based on its position would have to be identified and changed, leading to a potentially expensive and time-consuming maintenance burden.

Single versus multiple inheritance : Single inheritance derives from the classic tree hierarchy of classes, wherein each class has at most a single parent class. Multiple inheritance derives from the frame concept and argues that the tree structure is too limiting. Some objects can belong to more than one hierarchical structure.

Storage Management in Object-Oriented Systems: In a language having static binding, the system manages the storage. Languages using dynamic binding control the policies for its own use of storage. Smalltalk has methods for reclaiming space released by objects that no longer have references to them. In C++ and Objective-C implementations, storage management is left to the responsibility of the system or the application programmer. For large software systems, overlaying was a common technique but has not been used for sometime because memories are larger now and must realize hardware paging.

Pros and Cons of the Object-Oriented Approach: In this section, the main advantages and disadvantages of the object-oriented model as a programming methodology are briefly discussed.

Advantages: It has long been believed that object oriented system structuring offers advantages for design and implementation of computer software systems with respect to comprehensibility, verifiability, maintainability, etc. The major obstacle to widespread use of object-oriented systems has been its excessive execution overhead. Some authors argue that much of the execution overhead of object-oriented systems has been due to the implementation structure; that is, the objects are built upon a conventional layered software structure [Stroustrup 1986].

A wide range of benefits are claimed for the object model of programming, including increased modularity, separation of specification from implementation, and increased reusability of software components [Cox 1986]. The use of objects permits the programmer to manipulate data at a higher level of abstraction. Especially the display capability associated with objects is an attractive function for tracing and debugging in general. Object concepts can be implemented using non specialized languages, as in Lisp or Prolog [Wiederhold 1986].

Objects provide a useful abstraction in programming languages; views provide a similar abstraction in databases. Since databases provide for persistent and shared data storage, view concepts will avoid problems occurring when persistent objects are to be shared [Wiederhold 1986]. Object-oriented databases add characteristics such as persistence, concurrency control, resiliency, consistency, and the ability to query the database to facilitate access. Object capability has already been added to database management systems to provide for some of the benefits of encapsulation. Because, now, the user of the data is not protected from changes in representation and must know how to use the data [Dyke 1989]. The data can be temporarily converted into an object form while processing. Some authors claim that because of the ability to model very complex data and evolve the database without affecting the current application base, object-oriented database management systems will replace the relational data model implementations which emerged in the early 1980s [Davson 1989].

The concept of service can fit nicely into an object-oriented-framework. Object-oriented principles imply the concept of services. As a black box with a well-defined interface, an object is essentially a service access point.

Encapsulation provides the basis for building a system from modules that can be accessed through a well-defined interface. The abstract data-type approach defines the interface by a set of strongly typed operation (or method) signatures. Internal representation of the data types (by recording the methods only) can be changed without disturbing the rest of the system (interface with the object will remain the same). Encapsulation is a term that describes the scope of unrestricted reference to the attributes of an object. It may be desirable to restrict the freedom of other objects to retrieve or replace its attribute values. To provide access to attribute values, an object can provide access functions to allow other objects to inquire about its attribute values or to change them. Access functions control access to preserve privacy and integrity of attribute values by allowing them to be read and replaced only when appropriate. They can also provide traps to respond gracefully to ill-timed or ill-formed requests (semaphores).

Using the inheritance feature, type definitions can be related to each other through a type lattice. Type definitions can be incrementally modified by adding subtype definitions that change the original type. The combination of the supertype and the subtype produces a completely defined new generic type (dynamic type definition). The object-oriented data model has the ability to make references through an object's identity which is invariant across all possible modifications of the state of the object itself. This gives very much flexibility to change and modify the complex systems at very low maintenance cost [Davson 1989].

Persistence is an object's ability to outlive the process that created it. A persistent object exists in a memory space that is not dependent on any single computational entity, and a large number of objects (more than will fit into the virtual memory of a process) can be stored in this persistent memory space (i.e., the database) [Davson 1989].

Object-oriented programming languages help to manage related data having a complex structure by combining them into objects. An object instance is a collection of data elements and operation that is considered an entity. Objects are typed, and the format

and operations of an object instance are inherited from the object prototype (class). The prototype description for the object type is predefined and the object instances are instantiated as needed for the particular problem. The object prototype then provides a meta description, similar to a schema provided for a database. That description is fully accessible to the programmer internally, an object can have an arbitrary structure, and no user-visible join operations are required to bring data elements of one object instance together [Wiederhold 1986].

Dynamic binding allows storage to be defined at run time and is not unique to object-oriented programming. It allows location independent object invocation, and modification of internal implementation of objects without affecting the rest of the system.

Large programming projects are not infinitely divisible. One cannot reduce the elapsed time to completion by putting more people on the project. Object-oriented programming allows for a finer degree of subdivision, which is akin to the specialization of the workforce brought about by interchangeable parts. The concept of interchangeable parts had a profound effect on the ability to manufacture complex mechanisms [Dyke 1989]. To create self-contained component parts of a programming system and treat them independently is a goal similar to that of interchangeable parts. In object-oriented programming, each component is defined by its interface; and a complete description of the interface is to be all that is needed for using a component successfully. This provides a strong organizational reason for using object-oriented programming.

An object library can help to reuse the existing code. In doing so, new functions can borrow heavily on those that have preceded them, without the necessity of writing wholly new code. The hierarchical structure and inheritance capability allows for the creation of generic components that can be reused in many parts of the system.

Uniform interface between all objects, to system devices and resources can be provided by a common and uniform operation invocation mechanism. The designer of each component has the freedom to make internal changes that do not affect the interfaces, such as the internal representation of data (flexibility). Improvements that can be made within a class do not have to affect the users of that class. New classes and methods can

be added without affecting those already there, thus allowing for incremental modification (resulting in improved programming productivity).

Reliability of a physical system is unlikely to be proved formally, and a perfectly reliable system can not be achieved. Therefore, a designer must to be content with developing hardware and software techniques for improving reliability [Kohler 1981]. The embedded nature of most real-time command and control computer systems restricts (but not eliminates) opportunities for mounting determined attacks on the system. It is reasonable to provide a degree of assurance (at a moderate cost) that programming errors will not lead to serious system failures, by way of system-provided and enforced protection domains. In Alpha, for example, the kernel places each object in a separate domain, enforces this separation, and controls all interaction among objects and domains. Each object can invoke operations on only those objects for which it has explicit permission. By enforcing the separation of object protection domains, the general system objective of fault containment is advanced. In real-time system design, one should attempt to allocate resources judiciously to make certain that any critical timing constraint can be met with the available resources, assuming that the hardware/software functions correctly and the external environment does not stress the system beyond what it is designed to handle.

In distributed systems object replication can be used to enhance the degree of fault-tolerance and increase the availability of system services, and thereby support the predictability of the system performance much required in real-time applications.

Object-oriented programming is emerging as one way of using the continually improving capabilities of computer technology to provide improvements in programmer productivity and user function. Object-oriented programming is being used for a wide range of applications, particularly for knowledge-base systems involving close human interaction and judgment. It may also be used as a tool for specifying and building large, integrated data processing systems. As a vision and a set of programming techniques, object-oriented programming has great value in programming to build large systems that have long life cycles [Dyke 1989].

The aspect of operating system design that is most involved in meeting the needs of real-time applications is the manner in which contention for system resources is resolved, but

not in the inclusion or exclusion of specific functionality. Real-time operating systems should take into account timeliness constraints when resolving contention for resources. In Alpha, for example, all resource management decisions are based on the time constraints of the entity (thread/object) making the requests and when contention occurs, resources are allocated in a manner dictated by system's overload handling policies in order to maximize usefulness to the application (time-driven system resource management). The kernel's programming abstractions aid in the overall objective of global, dynamic, time-driven resolution of contention for system resources. The thread abstraction (as in the Alpha kernel) embodies a distinct run-time manifestation of logical computations which is more appropriate for distributed systems than the conventional process abstraction. This provides a direct means for associating the timeliness requirements that clients specify for their computations with specific location-transparent run-time entities that the kernel manages (a required feature to support graceful degradation in real-time systems). In this manner, global importance and urgency characteristics are propagated throughout the system along with threads, for use in resolving contention for system resources, such as processor cycles, communication bandwidth, memory space, or secondary storage, according to a client-defined policy. A more detailed elaboration of these issues can be found in the reference [Nortcutt 1987].

The object-oriented programming model used in Alpha is claimed to be especially well suited for the support of decentralized, high-concurrency implementations of the major reliability techniques for distributed systems (i.e. atomic transactions and replication), as well as modularity, information-hiding, and maintainability, normally associated with an object oriented programming paradigm.

Disadvantages: Object-oriented programming has however some drawbacks. It has been claimed to spend more processing resource handling message passing than would be required to perform function calls [Dyke 1989]. Moreover, it is acknowledged that the difference becomes minimal as the methods being performed become more complex (i.e., the granularity increases). For simple methods that must be performed repeatedly, it is possible to program a "fast path" around the message passing delays. Special hardware support for context switching of objects and message passing between objects can increase the general performance of object-oriented systems.

Object-oriented systems traditionally result in a workload that contains a great deal of context switching and data movement, thereby reducing performance. This increased workload arises because of the smaller granularity of system structures resulting from object-oriented design. Much data movement is among related object data structures. Some authors argue that, vertical partitioning lowers the flow of data across domain boundaries (intra-domain context switches are typically less expensive than inter-domain switches). Also, traditional object-oriented designs have not taken full advantage of semantic information available about the object's functions and storage. System structure affects the performance cost of satisfying inter-data and inter-functional dependencies and controlling the execution of a computation. The initial object-oriented systems (Smalltalk and Eden etc.) which have been embedded in conventional software environments (i.e., UNIX) have typically incurred increased execution costs (related to the control and recovery dependent performance overhead). A highly modular object-oriented program structure in the traditional implementation will incur an execution cost, which increases rapidly with degree of modularity because the separation of object defining and state management functions requires a large number of expensive context switches to execute state management and control functions.

It is claimed that the lack of efficiency of object-oriented systems is due to building objects on top of a traditional layered structure. Vertical programming within the objects encourage the use of the semantics by the type manager of each object. The keys to the performance improvement are type managers designed around data structures, compile time binding of functions to data, and the ability of type managers to use the object's semantic properties to optimize data processing, execution control, fault location, and recovery. Vertically partitioned object-oriented structures integrate implementation of type functionality and state management in a single execution context in order to reduce total cost. Vertical partitioning also encourages the exploitation of an object's individual semantic properties to minimize the execution of both the functions on data and system overhead functions, such as consistency management and recovery [Stroustrup 1986].

Whereas it is relatively easy to learn the basic concepts of object-oriented programming, it takes much longer for an individual to learn a large class library (this, of course, in the hands of the skilled user, might be a powerful programming resource) [Dyke 1989].

Static versus dynamic binding: Advocates of compile-time static binding say that it helps to discover errors at that time, and the program ultimately runs faster. Others say that run-time dynamic binding frees the developer from the constraints of having to make such fixed decision that may ultimately lead to more complex programs. A compromise position says that dynamic binding is best in the early development phases of a project, but that static binding is better later on, when the product is going to be installed for general use.

Despite the fact that dynamic generic types provide flexibility in development phase, it decreases the performance of overall system during execution (similar to that of dynamic binding vs. static binding).

Initial cost of developing an object model from scratch (if no object-oriented language initially exist) for use in lower system software levels is relatively pretty high, especially if the developed software system is not complex enough and its life cycle is not very long (in which case no need for much modularity, reusability, and maintainability).

3.4.2 Layered Model

In this section, a short evaluation of the layered model (which is chosen as the implementation methodology for the RT-DOS Kernel design) is presented, in comparison with the object model (with the reasons for which object model has been found unfeasible for the RT-DOS implementation). The main pros and cons of both approaches in comparison with each other are also given.

In a layered model, the operating system is organized as a hierarchy of layers, each one constructed upon the one below it [TanenbaumB 1987]. The first system formally constructed in this way was the THE system built by E.W.Dijkstra and his students [Dijkstra 1968], in 1968. A further generalization of the layering concept was present in the MULTICS system. But the layered model became popular especially after ISO OSI Reference Model for open systems interconnection was published by the International Standards Organization (ISO), as a set of protocols and communication services provided by the layers at each communication node specifying the network architecture. The term "open systems" refers to systems that can be interconnected to communicate with each other by conforming to common implementation standards. Here, the objective

is to guarantee the interoperability of products selected from different suppliers by virtue of the fact that they conform to a common suite of standards.

For building software architecture in general, the layered model is used to obtain modularity for reducing complexity and maintenance cost. In operating system kernel design, the layered model is used to separate basic kernel mechanisms (which are usually replicated on each system node) from optional upper level system services (which reside on nodes that need the related services). The driving principle used by the layered model is to keep the number of layers to a number that would make the task of describing and integrating the layers manageable. Layers can be partitioned into further sublayers as required. Though the ISO OSI is exactly a seven-layer model, a layered model used to build an operating system kernel might consist of a few basic layers to separate the functionality of the system into manageable and maintainable components; for example, TRON [Sakamura 1987].

Separating hardware-dependent parts of an operating system from other kernel components will provide portability, as well as hardware-independence which reduces software modification needs for each minor hardware change, such as adding a new hardware device or changing physical characteristics of an existing one. Most operating systems are designed as a collection of functional layers; for example, firmware layer, device drivers, kernel (I/O control unit, task manager, memory manager, etc.), OS policy layers (long-term task scheduling, system monitoring, file management, etc.), with applications running on top of operating system.

Though self-contained and independent modules of the layered model are easy to manage, the main drawback has been known as a performance decrease of the overall system in general. Vertical partitioning is claimed to be a solution which eliminates the performance problem, but it incurs implementation and maintenance costs due to its tightly-coupled functional modules. Nevertheless, the layered model is commonly accepted especially in communication systems area to establish very well defined standards between different software parties to increase interoperability between their products.

The main concern of the RT-DOS project team for choosing the layered model was to be able to separate minimum replicated kernel mechanisms from upper level policy services,

as well as to reduce the complexity of handling the project by separating it into self-contained and independently developed parts. A more detailed elaboration of the issue (reasons of the project team for rejecting the object model and accepting the layered) is presented in the following section (**3.4.3**) .

3.4.3 Object Oriented Approach Versus Layered Model

Although the RT-DOS project definition document [TayliA 1987] stated that the RT-DOS would be designed conforming to the layered reference model, it was felt to be necessary to explore the object oriented approach as an alternative design methodology early in the project. As noted above, a detailed survey [BorA 1989] has been carried out to investigate the relevance of the object model in the design of real-time distributed operating systems. The remainder of the section summarizes the conclusion which were reached.

In a detailed report [Davson 1989], the object-oriented paradigm was investigated as a potential alternative to the hierarchical layered system model. It is commonly believed that the modularity of object-oriented system structures leads to 'good' software systems which are comprehensible and maintainable. It is also commonly believed that these virtues can only be obtained on conventional computer architectures at the cost of performance and efficient use of resources. On the other hand, it is argued [Stroustrup 1986] that these inefficiencies and performance problems have historically arisen largely because past object-oriented systems have been built upon the inappropriate foundation provided by the conventional layered operating system and database system execution environment.

It is claimed that, a vertically partitioned structure for design and implementation of object-oriented systems demonstrates better performance of the application independent portion of the execution overheads in object-oriented systems, against the application independent overheads in conventionally organized systems built upon layered structures. It is argued that the gain in performance would be about 30-55% more than the conventional approach.

A non trivial distributed real-time case study [Stroustrup 1986] showed that vertically partitioned, object-oriented structured system results a 30-55 percent reduced overhead of context switches, CPU control flow and CPU data flow, over the functionally oriented

layered structured system. Evaluation of overhead was accomplished by implementing system control, but not implementing application functionality for both conventional layered and vertically partitioned versions of the application. The hypotheses that the case study group put forward are :

- Object-oriented systems implemented in the vertical partitioned framework are more efficient than a system of the same functionality implemented in the conventional layered system framework;
- That this system structuring technique yields a partitioning of state and functionality that results in effective and efficient distributed and parallel systems; and
- That this system structuring framework leads to verifiable robust systems with respect to fault tolerance.

The basis for the first claim is that special case state management algorithms are almost always more efficient than general algorithms, and that partitioning by type establishes a basis for locality and computational operations. Both type specific functions and state maintenance functions can take advantage of the semantics of the typed objects. All mechanisms for monitoring integrity in the presence of faults are based upon redundancy. Semantic properties of data and data structures can be used to improve overall system performance. Type manager implementation of functions often allows compile-time binding of consistency and fault management functions to their associated data structures, instead of run-time interpretation of generic operating system or database system functions.

Compile-time binding typically results in improved system efficiency. Decomposition of total processing on the basis of application-based structures cuts down the data flow and inter-domain context switches. Performance may also be improved due to the locality of functions and data within the object boundaries.

Simplicity of recovery results from atomicity at object boundaries and from use of type-specific fault detection and recovery mechanism. Vertical partitioning limits the propagation of faults, localizes data, limits the interdependencies among objects for fault recovery and enhances overall system security.

In message-based systems, the importance of a computation is lost as messages are sent to different servers as the computation progresses. Each step of a computation is performed at the priority of the server, not the priority of the computation itself. The priority of a message can be used to resolve conflicts for communication resources. To resolve contention for all types of resources, computations have to retain their priority across all the steps. This would require that server processes endorse the priority of each message they receive. Furthermore, server processes must be preemptable so that when a message with higher priority arrives, server suspends its current work and initiates the new request. Thus, the thread/object approach of object model adopted in Alpha seems to be a promising approach to effectively solve the contention problems introduced by distributed real-time tasks.

On the other hand, requirements of real-time systems necessitate exploitation of the particularities of underlying hardware, considering the efficiency above all other criteria. Nevertheless, in theoretical studies and emulations of operating systems most of the interactions with the hardware are omitted and many low-level issues discarded as "engineering and technology dependent details". This simplistic approach harms the conceptual nature of the operating system. Such details have substantial effects on the final implementation, as some low-level mechanisms often conflict with the assumptions of the high-level design. For instance, in the Alpha implementation, a simple change in the version of CPUs of application processors (from M68010 to M68020) resulted in a complete revision of all memory management and protection policies (hence changes in object and thread structures, and context management). The upgrade, which tremendously improved object context switching, has been achieved at the cost of modifying most of the kernel code.

Though the object model proves to be a suitable framework for the design of real-time distributed systems, its adoption for the RT-DOS suffers from :

- The absence of proper hardware support;
- Unavailability of object-oriented development tools; and
- The cost of initial investment to build basic abstractions of the object model.

The Transputer architecture does not allow a straightforward implementation of the object model. Transputers support neither virtual memory nor any kind of protection mechanism much needed to create disjoint protected object contexts. The alternative of

confining one object per Transputer would restrict dynamic creation of objects, and cause a decrease in system efficiency (not tolerable in real-time systems). The lack of hardware support precludes effective implementation of fault-containment mechanisms, which minimizes the benefits of the object model.

At the present, Transputer development platforms do not offer object based design environments. Even if such a system existed, the programming tools would probably incur additional overheads, decreasing the predictability of the final real-time system.

Furthermore, the cost of the building an object-oriented development platform from scratch is tremendous. In the Alpha experience, building basic object abstractions and implementing some part of the kernel mechanisms on top of these abstractions cost about 30.000 lines of C. This sizable number represents an amount of investment which is beyond the limits of the RT-DOS project.

3.5 General Survey on Related Work Areas

In this section, a number of DOS kernels are presented as the current approaches to kernel design issue. A detailed report covering the survey study on related work area, carried out by the RT-DOS project team members, has been published [TayliB 1990]. The interprocess communication mechanisms of these DOS kernels are discussed in detail in **Chapter 5**, and hence will not be covered here.

In the preliminary design of the RT-DOS Kernel, the architecture of a large number of distributed and real-time systems was surveyed. Those found of particular interest to the RT-DOS project have been studied in detail with respect to their process model, interprocess communication model, and other subjects, such as process migration support and predictability of provided services. The main features of different architectures are exemplified in the following systems:

- *ACCENT* [RashidB 1981];
- *AMOEBA* [TanenbaumC 1981];
- *CHARLOTTE* [ArtsyB 1989];
- *DEMOS/MP* [Powell 1983];
- *LOCUS* [Walker 1983];
- *SPRING* [StankovicB 1989];

- *V Kernel* [CheritonA 1988];
- *ALPHA Kernel* [Nortcutt 1987];
- *ARTS* [HideyukiB 1987];
- *HELIOS* [Helios 1989].

The objective of the Alpha Kernel project was to investigate the relevance of the object model in the design and implementation of real-time distributed systems. As a comprehensive coverage of the Alpha Kernel has been provided in a detailed report [BorA 1989], and has been explained in different sections of the previous chapters, it is not discussed here any further. Problems related to the design of IPC model, process model, and process migration mechanisms; as well as design alternatives, solutions adapted in the surveyed systems, are presented in **Chapter 4** and partially in **Chapter 5**, along with their rational. Only some of these surveyed kernels are discussed below, briefly .

ACCENT: It was designed to support a distributed project and development of a fault tolerant distributed sensor network. Its philosophy is uniform and transparent access to distributed resources through message interfaces for loosely coupled uni/multi processor systems. It implements a single, powerful abstraction of communication between processes, with all kernel functions accessed through messages. It provides location independent naming schemes, and implements process migration for fault tolerance. It provides rapid error detection and tools for transparent fault-recovery, debugging and monitoring, as well as optimizing file access and providing IPC mechanisms through VM support.

Accent can be viewed as a loosely connected collection of host machines. Each host machine on the network is equipped with an OS kernel which in turn supports a collection of processes. Functions provided by the system kernel include:

- Interprocess communication;
- Virtual memory management;
- Process management;
- Access to devices through the IPC;
- Support of application specific micro code;
- Support for process monitoring and debugging.

Services, referred as a set of commands and responses, are implemented by one or more server processes. The number of servers providing a service is transparent to users of this service. All communication between a process and the kernel is done through a special channel. It is possible for a process *A* to set its kernel port to another process *B*. This mechanism forms the basis for remote debugging and monitoring systems.

Basic primitives allow create, destroy, monitor (status), suspend and resume operations on a process. All objects (opened files, virtual terminals, etc.) and services (process creation, I/O, etc.) in Accent are accessible through the IPC. Even though the IPC facility of the kernel is defined solely in terms of communication between processes on the same machine, communication can be extended transparently over the network by processes called network servers.

The basic transport abstraction of the IPC is the notion of a PORT. A port is a protected kernel object associated with an individual queue. Ports are created by a system call (*AllocatePort*), and initially owned by the creating process. The result of a port creation call is a local port name (a capability), used as an index into a correspondence table, maintained by the kernel for each process. Whenever a port name is passed in a message the system kernel must map that name from the local name space of sending process into the name space of the receiving process. The ability to manipulate access to ports allows for the redirection of communication from one process to another and the explicit management of communication between two processes by a third process.

Subject to implementation restrictions on maximum port size, the process with receive rights is allowed to specify the maximum number of messages which may be queued for that port at a time. An attempt to send more messages results in one of the following depending on the options specified by the sender:

- Sending process is suspended until the message can be placed in the queue (per process-system server relationship);
- Sending process is notified of an error condition perhaps after a time out (likely the case of a process checking the status-awake, dormant, etc.- of a port);
- Kernel may accept one message per sending process to be queued later (case of a server process finishing a transaction). The server can't afford to be suspended waiting on a user to clear its queue.

Accent provides the tools for migrating processes since all relevant process state (micro state, port queues, memory) can be extracted and transmitted over the network. When migration occurs due to hardware problems, higher level mechanisms are required for fault tolerance. In such a case the migrated process state probably corresponds to the last "good", checkpointed state, and some communication may have gone on since it was saved. The issue of process migration for fault tolerance is not only a kernel problem but also one of properly structuring message communication into atomic transactions.

AMOEBA: Amoeba is a distributed operating system running on a collection of Motorola, Intel, and VAX/PDP 11 based machines which are connected through a LAN. Its objective is to control a collection of machines based on a processor pool model. It implements process migration for load balancing, and the process migration mechanism is similar to the process creation. Task creation is done through a server process, rather than by kernel calls. It has UNIX-like naming mechanisms, and file structuring.

Like Accent, the AMOEBA services are implemented as a set of commands and responses, on one or more server processes. The number of servers per service is transparent to users of this service. Services are of two category (the system does not differentiate between them):

- Public services, such as disk I/O, file access, directory, database management, etc. They are carried out by long life span processes;
- Private services, such as short lived processes, started up to run a specific program for a specific user.

Each AMOEBA machine runs a resource manager process that controls that machine. This process belongs to the kernel for efficiency reasons. Key operations it supports are create segment, write segment, read segment, and make process.

To create a process, the parent executes "create segment" for text, data and stack, getting a capability for (a pointer to) each segment. It then fills each one in with that segment's initial data. Finally it performs make process with these capabilities as parameters, getting back a capability for the new process. The parent builds a process descriptor consisting of: required CPU type, port from which the process code can be fetched, command string, argument string, environment string, umbilical port to send

exit status to the parent, inherited ports. Using these primitives, it is easy to build a set of processes that share text and/or data segments.

Interprocess communication is based on the datagram services exchanged through PORTS. The datagram model is used to reduce the system's complexity and give processes the flexibility to define their own protocols and flow control scheme. Higher level protocols ensure reliability of communication primitives. Messages are unbuffered, and basic primitives are synchronous with the provision of an exception request which is propagated recursively through a hierarchy of servers.

Ports are used like UNIX file descriptors. They can be inherited and passed around the same way as in UNIX. Ports are stored in directories like i-nodes, and two category of port exist for : public servers (known and/or publicized names), and private processes (secret names).

Processes may migrate during the execution to more appropriate systems which were not available during the creation of the process (providing that two systems have compatible CPU architectures). The migration mechanism is very similar to process creation mechanism. When a monitor wants to move a process, it creates a port (file) from which the process code can be fetched, and builds a descriptor which is sent out to look for a new site. When some system has inherited the process, the originator may discard it.

The basic idea behind fault tolerance in AMOEBA is that machine crashes are infrequent, and that most users are not willing to pay a penalty in performance in order to make crashes 100 percent transparent. Instead, Amoeba provides a boot service with which servers can register. The boot service polls each registered server at agreed intervals. If the server does not reply properly within a specified time, the boot service declares the server to be broken and requests the process server to start up a new copy of the server on one of the pool processors. As the Amoeba IPC is not based on virtual circuit or session concept, the change of the server is completely transparent for users if their demands can be served by any one of the similar servers.

CHARLOTTE: is a distributed operating system which has been developed on VAX 11/7xx systems connected through a Pronet Token Ring network system. Its objective was to serve as a testbed to evaluate distributed algorithms; to support experimentation

with policies for load sharing and balancing; and to explore the design issues that process migration raises in a message based system.

Its replicated kernel provides the IPC and short term scheduling. It emphasizes policy/mechanism separation of the kernel to support task migration. No information is retained at the source kernel, related to the migrant process.

Each machine in the Charlotte system runs a kernel responsible for simple short-term scheduling and message-based interprocess communication protocol. Processes are not swapped to the backing store. A battery of privileged processes, called utilities, runs at the user level to provide additional OS services and policies. The kernel and some utilities are multithreaded.

Charlotte kernels collect statistics on the machine load (number of processes, links, CPU and network loads), individual processes (age, state, CPU utilization, communication rate), and the most active links. The kernel periodically summarizes sample data and delivers it to policy making components.

Processes communicate via location independent links, which are capabilities for duplex communication channels. Charlotte IPC allows concurrent communication over multiple links, selectivity, message cancellation, and link transfer. When moving processes, the communication is suspended until the operation is completed. Outgoing messages from the moving processes are buffered in the process's virtual address space, and incoming ones are buffered in their sender's virtual address space. Since Charlotte provides message caching, no unreceived incoming messages from the moving processes are lost.

Charlotte kernels are responsible for migration mechanisms (to detach, transfer and reattach migrant process) and for providing policy making services with time and load statistics. Four service calls have been added to the process-kernel interface to implement process migration. These are:

- Start/stop gathering statistics;
- Migrate a particular process to a particular machine;
- Accept a process from a particular machine and allocate memory to it;
- Cancel a migration in progress, if possible.

Transferring processes occurs in three main phases : negotiation, transfer, and establishment. After that, policy making components in the source and destination machines are informed that the migration has been successfully completed.

DEMOS/MP: DEMOS/MP is a distributed operating system running on Z8000-based machines connected through a LAN. Its objective is to allow users to access the multi-processor system in the same manner as a uniprocessor system. In DEMOS/MP, process control functions are performed without concern of where processes are actually located. Processes' state are well encapsulated, minimizing migration costs, and it implements process migration for load balancing and fault tolerance.

DEMOS/MP is a message based OS. Most of the system functions are implemented in server processes which are accessed through the communication mechanism. A copy of the kernel, which implements the basic objects (processes, the IPC, links), resides on each processor. Although each kernel independently maintains its own resources, all kernels cooperate in providing a location independent and reliable IPC.

DEMOS/MP processes consists of :

- Switchboard, which is the server that distributes links by name;
- Process manager, which handles high level scheduling decisions for processes. It allocates, keeps track of resource utilization and monitors processes' state through messages to concerned kernels. For instance, it may decide when and where to migrate a process;
- Memory scheduler, which offers similar functions to the process manager;
- File system (implemented with four processes);
- Command interpreter, which forms DEMOS user interface.

In DEMOS/MP, messages are sent using links to specify the receiver of the message. Links can be thought of as buffered, one way message channels, but are essentially protected global process addresses accessed via a local name space. Links are created, duplicated, passed to other processes, or destroyed through kernel operations. Addresses in links are context independent; if a link is passed to a different process it still points to the same destination. A link may also point to the kernel which interacts with user processes in the same manner as ordinary processes.

The most important part of a link is its message process address. This is the field which specifies to which process messages sent over the link are delivered. It consists of a fixed part with a unique global process identifier, and a varying part which is the last known location of the process.

Since DEMOS/MP guarantees message delivery, in moving a process it must be ensured that all pending and future messages arrive at process's new location. There are three cases to consider:

- Messages sent but not received before the process finished moving;
- Messages sent after migration using an old link;
- Messages sent to a new link.

Messages in the first category are moved with process context to the new site. The third case is trivial. The middle case requires a follow up procedure.

After a process is moved a forwarding address is left at the source site. When a message is received at a given site, if the receiver is a forwarding address, then the location address part of the incoming message is updated and the message is resubmitted to the system. The forwarding mechanism is a sufficient provision to ensure the proper and continuous operation of the IPC. However, performance considerations requires gradual restoration of correct link addresses. As link information is disseminated overall network, it is not feasible to try to update them all at once. Instead, as messages are received by the old site and forwarded, the old kernel can notify the originator's kernel of the address change and avoid subsequent rerouting process. The forwarding address is needed as long as old references still exist, or the migrated process is alive. Its removal can be an event based process or garbage collection activity.

A number of DEMOS/MP design features have made the implementation of process migration possible. DEMOS/MP provides a complete encapsulation of a process. There is no uncontrolled sharing memory, all contact with the OS and other processes is made through the links. There is no process state hidden in various modules of the OS. The location transparency and context independence of links make it possible for both the moved process and processes communicating with it to be isolated from the change in venue.

LOCUS: Locus has been implemented on VAX machines connected through an Ethernet LAN, as a distributed operating system. Its main objectives were transparent access to network data, transparent remote execution, high reliability, and UNIX compatibility. Abstraction of physical boundaries, even across heterogeneous CPUs, was the another goal to be reached. In Locus, the main highlights are:

- Network wide tree structured naming hierarchy;
- High performance in accessing remote objects;
- Support of nested transaction;
- Partitioned operation of subnets and their dynamic merge;
- Replication for increased availability, performance, and fault tolerance;
- Process migration.

Locus is a procedure based OS. Processes request system services by executing system calls, which trap to the kernel. The kernel runs as an extension to the process and can sleep on its behalf. Distributed nature of the system is hidden from higher levels. When the assistance of a foreign service is needed, the OS packages up a message and sends it to the relevant site. Typically the kernel then sleeps, waiting for response, much as it would after requesting a disk I/O on behalf of a specific process. This flow of control is a case of remote procedure calls.

Transparent support for remote process execution requires a facility to create a process on a remote machine, initialize it properly, support cross machine, interprocess functions with the same semantics as were available on a single machine, and reflect error conditions across machine boundaries.

In Locus, as in UNIX, the name catalog also includes objects other than files; devices and IPC channels are the best known. An IPC model is often a controversial issue in a single machine OS, with many differing opinions for possible implementations. In a distributed environment, the need to support of error handling imposes a number of additional requirements that help make design decisions, potentially easing disagreements. In Locus, the initial IPC effort was further simplified by the desire to provide a network-wide IPC facility which is fully compatible with single machine functions that were already present in UNIX. Therefore, UNIX named pipes and signals are supported across the network. Their semantics in Locus are identical to those seen on

a single machine UNIX system, even when processes reside on different machines in Locus.

Processes are typically created by the standard UNIX fork and exec calls. The decision about where the new process is to execute is specified by information associated with the calling process (set explicitly or implicitly). In both cases, a process body is allocated at the destination site, and common information (open file descriptors, parent's state information, etc.) are to be properly initialized. In the case of "fork", the process address space both code and data must be made a copy of the parents'. Ideally, if the code is reentrant and a copy already exists on the destination machine, it should be used.

The major difficulty in initiating remote actions resides in the semantics of the objects which assume memory sharing. In Locus, a token mechanism, which marks which copy of a resource is valid, is used to address such issues. In the worst case, performance is limited by the speed at which the tokens and their associated resources can be flipped back and forth among processes on different machines. Such extreme behavior is exceedingly rare. Virtually all processes read and write substantial amounts of data per system call. As a result, most collections of UNIX processes designed to execute on a single machine run very well when distributed on Locus.

V KERNEL: It is a distributed operating system running on SUN Workstations connected through an Ethernet LAN. It was designed to check the feasibility of building a distributed system with all network communication using the V message facility, even when most of the nodes have no local storage. Its philosophy was to provide a software backplane; that is, a relatively small OS kernel which can implement network transparent abstraction of address spaces, lightweight processes and the IPC. The rest of the system can then be built at the process level in a machine and network independent fashion.

V is a message oriented kernel that provides uniform local and remote IPC. In the V kernel, handling shared state- shared memory can be implemented across multiple machines. It supports group communication (multicasting) as opposed to broadcasting.

V kernel provides time, process, memory, communication, and device management services. Each of these functions is implemented by a separate kernel module and

replicated in each host. Each module is registered by the IPC and used with the same IPC interface as user processes. For example, a new process is created by sending a request to the kernel process server, and a new address space by sending a request to the kernel memory server.

The kernel process server implements operations to create, destroy, query, modify and migrate processes. As opposed to conventional operating systems, the V kernel minimizes the process management mechanisms as follows:

- Process initiation is separated from address space creation and initiation;
- Process termination is simplified because there are few resources at kernel level to be claimed. Most resources are managed at process level. Moreover, the kernel makes no effort to inform servers when a process terminates. Each server checks periodically its clients, reclaiming the resource if the client no longer exists;
- Scheduling is simplified by the kernel providing only simple priority-based scheduling. A second level of scheduling is performed outside the kernel by scheduler processes;
- To avoid the full complexity of exception handling in the kernel, the process management module simply causes the exception-incurring process to send a message describing its problem to the exception handler server. The exception server then takes over, using the facilities of the kernel or other higher level servers to deal with the process (i.e., invoking an interactive debugger).

Processes are organized into groups called teams. A team of processes share a common address space, and therefore must all run on the same processor. A process is identified by a 32-bit globally unique process identifier. The high order 16 bits serve as logical host identifier while the low order 16 bits are used as locally unique identifier. In Ethernet the top 8 bits in the logical host identifier are the physical network address of the workstation, making mapping of the process identifier to network address trivial. At the next level, services can have symbolic names in addition to their *pids*. A service can register a symbolic name with its kernel so that clients can locate, through broadcast queries, the service by name.

The kernel IPC facility was designed to provide fast transport level services. Communication between processes is provided in the form of short fixed length

messages, each with an associated reply message, plus a data transfer option to move larger amounts. The request/answer model provided a schema which can then be implemented according to the message model or procedure call semantics.

The VMTP protocol [CheritonC 1986] supports datagrams, multicast, forwarding, streaming, security and priority. It is optimized for request-answer behavior. In particular, there is no explicit connection setup or tear down. The kernel is structured to minimize the cost of communication operations. Every process descriptor contains a template VMTP header, initialized at process creation. Using this header, the overhead of preparing a packet is reduced. In particular, there is no need to allocate a descriptor buffer for queuing.

Process migration was retrofitted into V. The ability to extract context information, to freeze, and unfreeze processes was added to system primitives, later. Process migration suffered from the original definition of process id which included a physical reference. The "logical host" subfield is then eliminated from the process id to facilitate the process migration.

Since it was designed primarily for use in an interactive environment, V support for fault tolerance is minimal. The kernel detecting an error sends a specially formatted message to the exception server, which is outside the kernel. The exception server then invokes a debugger to take over. This schema does not require a process to make any advance preparation for being debugged and in principle, can allow the process to continue execution afterwards.

All these past experiences of currently implemented distributed operating systems have been taken into consideration carefully in the design of the RT-DOS kernel architecture. In the following chapter (**Chapter 4**), the RT-DOS kernel architecture is presented with the influences of the aforementioned kernels on its design.

4. THE RT-DOS KERNEL ARCHITECTURE

In **Chapters 2** and **3**, an overall view of real-time distributed operating systems' architectures and their basic concepts, have been presented in general. In this chapter, the design considerations of the RT-DOS Kernel, its basic data and control structures, as well as logical and physical models used for the implementation, are discussed.

4.1 Design Considerations of The RT-DOS Kernel

The RT-DOS Kernel, replicated at each node as a collection of self-contained mechanisms, provides the minimum services, including process management, memory management, interprocess communication (IPC), global time management, short-term scheduling, and process migration, to the upper layers.

The RT-DOS Kernel was planned to be a fully distributed, message based system. It was designed to provide a dynamically reconfigurable work platform:

- To adapt to dynamic changes in workloads (in the prospect to meet prescribed execution deadlines);
- To allow system maintainability, and dynamic upgrading;
- To increase system reliability and availability (fault tolerance and graceful degradation are supported through redundancy of interchangeable resources and software-based mechanisms such as process migration or replication).

The RT-DOS modules and services are designed to be scattered throughout processing components. Their location and number is intended to be dynamic and transparent to users. The RT-DOS Kernel hides the existence of multiple processing units and network interconnection from the process level, and provides a unique system abstraction.

Design of a real-time distributed computer system, based on a network of Transputers, raises the following sequence of questions:

- How can real-time applications be defined, configured to run on loosely-coupled processing units, and how can their performance be accurately predicted ?
- What should be the basic constituents of the distributed system supporting real-time applications ?
- What should be the topology of the underlying network of Transputers ?

The first question relates to the definition and implementation of real-time applications, and this issue has been elaborated in **Chapter 2.** The second and third questions concern respectively the logical and physical models of the RT-DOS Kernel, which are among the subjects of this chapter.

The preliminary study of the kernel has clearly shown that its functions can and should be separated into two sublayers based on the desirable separation between mechanisms and policies which is introduced in **Chapter 2.** A first sublayer implements necessary mechanisms constituting a micro kernel, and basic services built on top of it. In the design of the kernel, the first mechanism-oriented sublayer is seen as "domain-independent" in the sense that it implements mechanisms common to various types of operating systems. This first sublayer is presented in the paper [TayliD 1990]. The second sublayer is considered as "domain specific" and as such, implements intrinsic features of the target systems. In the RT-DOS Kernel project, this sublayer implements real-time features of the system, such as predictability, flexibility (dynamic reconfiguration), dynamic load balancing, and fault-tolerance. This approach enforces the view that the main difference between various categories of operating systems resides much more in the policy-making parts (particularly in the resource scheduling) than in the mechanisms necessary to implement these policies. The separation between policies and mechanisms is known to be important in integrating dynamic aspects of modern real-time systems [TatliB 1990, StankovicE 1988].

The IPC mechanism which is the subject of this thesis, is a vital part of the micro kernel (first sublayer), and provides a message based communication service for the rest of the kernel components, including the second sublayer. The rest of the chapter is devoted to the concepts that helps to distinguish between these kernel layers, putting emphasis on the mechanisms.

4.2 Granularity of Process Objects

Design of the process model starts with the investigation of the granularity of processes, naming schema, concurrency control/coordination model, and process context. The definition of the RT-DOS process model is based on the answers to these questions.

Executing entities on operating systems are conventionally modeled with two complementary objects : *Processes* (*Tasks*) and *Lightweight Processes*. The term process is used in this thesis in its usual meaning to define an operating system entity that has its own state, address space and context, while task will refer to the basic constituents modeling concurrent and/or sequential activities of real-time applications. Lightweight process denotes those entities that do not carry the weight of a separate address space, and context. That is, there can be multiple lightweight processes per address space, sharing the same context. The term thread is also used interchangeably with lightweight process.

The need for lightweight processes, to build parallel execution paths besides the processes, is rather an optimization issue. Their usage can be justified in a number of ways :

a) Most of the programming languages which offer parallel constructions suffer from the high overhead incurred by process creation/destruction mechanisms. In fact, conventional implementations use the process object to materialize both independent computation units (i.e., jobs of two different users) and closely related actions (i.e., two instructions to be run in parallel in a given program - like OCCAM PAR construct [Geraint 1987, Pountain 1987]). While the cost of creating a heavy structure including address space, resource allocation, accounting information may be justified in the first case, the second suffers from unnecessary investment of duplicating system information. As a result, many parallel constructs in programming languages become ineffective, hence obsolete due to the inappropriate granularity of the process object;

b) Given the need for extra provisions in protection and process management mechanisms, not all operating systems offer multitasking facilities. Nevertheless, they had to provide substitute services to support some level of concurrency. The common approach is the implementation of a dual system interface. System calls, which require the suspension of process execution for some time, are presented in two versions:

   - synchronous (blocking) calls; and
   - asynchronous (non-blocking) calls.

   Users which are keen to proceed with a parallel activity, while a blocking action is expected, invoke an asynchronous version of system calls. They check later on

the outcome of the requested service using other primitives. Although this dual interface approach solves somehow the concurrency needs, it results in a number of major drawbacks. First, the system needs to present additional interfaces increasing its size and decreasing its intelligibility. Further, the consistency of requested system actions is left to the control of the application, since the operating system can't check their logic. Finally, given both facts, the reliability of applications built with dual systems interfaces is rather diminished.

The addition of lightweight processes (threads) to the process model provides an adequate solution for the above problems. If the cost of starting a new thread of execution is negligible, blocking actions can be initiated as parallel flows, while the execution may proceed following concurrent threads. Further, all system services can be provided as synchronous requests. This reduces the overall investment, number and complexity of system calls, and increases system performance and reliability.

The RT-DOS process model supports both the process and lightweight process (thread) objects. Processes form conventional building blocks of the operating system to which system resources are assigned. Applications defined as a network of tasks (or task force) are represented in the operating system by an equivalent network of processes, which in turn is mapped on a network of processors based on time and resource constraints. Threads represent in the RT-DOS, parallel execution units within processes. Their existence is revealed to the operating system only by the process scheduling queues. They use and share resources allocated to the encapsulating process and acquire new resources on behalf of it. Unlike processes, threads can access common memory areas, consistency of shared objects being guaranteed by programming constructs. Like processes, threads communicate with each other through the low cost RT-DOS interprocess communication (IPC) mechanisms.

4.3 Process Naming Schema

Naming is a problem of mapping between domains [Watson 1988]. Mapping may involve many levels. For example, to use some service a process might first have to map the service name (generic name) onto the name of a server process which may not be unique. As a second step, the reference of the server is to be mapped onto a specific processor.

Process identifiers follow the same semantics as other system objects such as files, directories, communication ports, I/O devices etc. Therefore, similar policies can be applied for their generation with their pros and cons. For example, *V Kernel* uses a global (flat) naming space for identifying processes, while *LOCUS* applies a network wide tree structured hierarchy. Unique names may include location dependent references [CheritonA 1988], as opposed to location independent naming [CheritonA 1988]. Another variant is to use a level of indirection through local references as used in DEMOS/MP and ACCENT for communication entities.

A global naming policy is a basic requirement of distributed systems. However, the location transparency is a debatable issue. The use of an explicit host field in a process identifier allows distributed generation of unique process identifiers, and an efficient mapping from process id to network address. In particular, it is very efficient to determine whether a process name renders the process migration impossible.

The architecture of the RT-DOS and the granularity of its components suggest the adoption of a global and flat process name space. This issue, closely related to the naming of communication elements, should be handled along with the IPC naming schema.

4.4 Concurrency Control, Coordination, and Monitoring

Processors in loosely coupled distributed systems do not share primary memory, and so synchronization via shared memory techniques such as semaphores and monitors is not generally applicable. Moreover, memory sharing is considered as an artificial technique in loosely coupled systems if ever implemented. Despite its attraction in optimizing the performance of local access mechanisms, shared objects reduce system modularity, disseminate process dependent information through the system, increase the cost of process management (monitoring, migration, etc.).

Message-based primitives constitute a natural coordination and communication model for fully distributed systems. They help to encapsulate the information and reduce the location dependency. However, the overhead of message-based primitives is generally

high, especially when addressing local objects. Therefore, extreme care must be taken to enhance the cost/generality ratio.

A closely related issue is process monitoring. An operating system element (scheduler, debugger, etc.) or a user task may wish to control the computation of a given process. A versatile operating system kernel has to make provision for such a service. The majority of the systems use a static parent-child process connection to implement this mechanism. DEMOS/MP adopted an elegant solution in which messages forwarded to the system kernel are tagged with a special indicator which is intercepted by the kernel of the system on which the process executes.

The RT-DOS is a message-based system, so that concurrency control and process coordination mechanisms are expected to be implemented with the IPC mechanisms.

4.5 Process Migration versus Replication for Fault Tolerance and Load Balancing

Both process migration and replication of services (and/or data) are used to increase availability of services, to improve fault tolerance, to better utilization of specific services, and finally to provide dynamic/static load balancing.

Process migration is the relocation of a process, by transferring a sufficient amount of its context, from the processor on which it is executing to another processor. Process migration is normally an involuntary operation that may be initiated without the knowledge of the running process or any processes interacting with it. Ideally, all processes continue execution with no apparent changes in their computation or communication.

Process migration has been proposed as a versatile solution in a number of basic mechanisms which implement the adaptability, transparency, fault-tolerance, increased performance in distributed systems.

The overall performance of a distributed system can be increased by distributing the load among available components of the system. However, even a carefully planned system is subject to unpredictable internal or external events which may result in a bottleneck and

be subject to overheads. Besides this global optimization, often a small subset of the processes running on a distributed system account for much of the load. Some effort spent off-loading such processes may yield a big gain in performance [Leland 1986]. Such resource load balancing is impossible to achieve with static assignment of processes to processors. If it is possible to monitor the evolution of the processing in a system and adapt the distribution of resources to changing needs, a system has better chance to increase its throughput and meet its scheduled deadlines, despite the load incurred by these extra activities [Livny 1982, Eager 1986].

In a distributed system, communication costs are often accountable for the degradation of the overall performance. There are instances where moving a process nearby a frequently used server (a database or file server to a nearby server with the same functionality), or just clustering communicating processes closely may yield substantial gains [Lo 1989]. Process migration can also be useful in taking advantage of some special purpose hardware device, needed during a given slot in the lifetime of a process.

For such reasons as shut down for maintenance, reconfiguration, upgrading etc., a processor or subsystem has to be off-loaded and the entire execution context transferred elsewhere to ensure the graceful degradation of the system. If the information necessary to transfer a process is saved in a stable storage, it may be possible to transfer a process from a crashed site to a working one (fault recovery).

The design of process migration requires answers to the following questions:

- What is the minimal context to be transferred ?
- How can process state be encapsulated ?
- How can process communication links be moved ?
- How can a hanging communication be discovered ?
- How long does communication continue in a migrating process ?
- Where and how to dissociate migration policies from mechanisms ?
- What is the impact of the process migration on the predictability of real-time applications ?

The process migration can be simplified if it is assumed that:

- Kernel capabilities should be compatible throughout the system (all kernels providing similar services);

- Process state should be well encapsulated (kept within process context as much as possible);
- Processes should interact with each other through regular interfaces, i.e., messages (no uncontrolled access to system resources and other process' environment);
- Objects in processes should be referred to with location independent identifiers (process itself, ports etc.).

Replication of system services or shared data at different nodes of the network increases the availability of these resources to user processes, while decreasing the communication traffic and providing better concurrency by reducing exclusive access to resources. On the other hand, keeping replicated copies of shared data up-to-date is costly, and might increase the network traffic because of update synchronization procedures.

A replication mechanism can be used basically to increase the availability of resources, while process migration is used for dynamic load balancing. Both mechanisms can be used to increase fault tolerance, though process migration provides more graceful degradation in case of failures.

The RT-DOS Kernel is designed to provide the upper levels with the basic mechanisms on which fault-tolerant applications can be built. Implementation of both process migration and replication techniques are either directly related to the IPC mechanism or severely affect the design of it. In fact, the IPC mechanism of the RT-DOS Kernel has been designed with these considerations are in mind. The multicasting feature of the IPC mechanism supports data and process replication, while keeping minimum state information about the communication entities (links, addresses, processes) should eliminate problems during migration of processes between nodes dynamically. These issues are discussed in the next chapter (**Chapter 5**) in more detail.

4.6 Resource Management (Process Allocation and Dynamic Reconfiguration)

Resource management is the concern of policy making components of an operating system. Hence, the topic is beyond the scope of this thesis. However, some view is still necessary given the close relationship of this issue with those already introduced.

The main problem with managing resources in distributed systems is the lack of accurate global state information. Management decisions are taken in uncertainty, feedback information is delayed, and often obsolete. However, as the target of the RT-DOS is real-time applications and the system is implemented on loosely coupled low-granularity modules (Transputers) the only critical resource seems to be the processing unit (along with the communication bandwidth of the serial links). Resource management is related to application configuration (static allocation of processors), monitoring of resource utilization, and dynamic reconfiguration of the system.

Processor Allocation : The processor allocation problem can be summarized as the mapping of a network of processes on a network of processors. A common approach is the hierarchical structuring of processing units. For each group of worker processors, a manager is assigned to monitor the evolution of the processing. A similar schema can be applied to other concurrent applications. Managers can also be organized hierarchically. They report to a higher level authority. This hierarchy can be extended with as many layers as needed. However, worker-manager-president-etc., relationship is rather unreliable structure. If the hierarchy is broken at the top level, the system stops functioning. To avoid having a single, thus vulnerable decision maker at the top of the tree, one can decide to substitute this level with a ruling committee. The hierarchical allocation schema, with all its variants, does not conform fully to the idea of distribution. More suitable organizations can be formed by assigning horizontal responsibilities, i.e., managers communicating with each other to coordinate their activities, in addition to their local duties.

Dynamic Reconfiguration : Application configuration correspond to the initial mapping of the processes onto the processing units. The configuration process is expected to secure enough resources for proper execution of the application. Yet, the dynamics of the execution may cause unpredictable delays in communications and change resource requirements of the applications. Permanent monitoring is then a necessity with the prospect of dynamic reorganization in resource assignments. Increasing parallelism, minimizing interprocess communication costs and balancing the load are among the criteria leading to dynamic reconfiguration. Real-time applications often run well specified and carefully tuned configurations. Dynamic behavior of the application is expected not to exceed prescribed limits. Nevertheless, any system is still prone to

unpredictable errors, and is expected to react to such changes (with a revised global resource allocation). The RT-DOS Kernel has been designed to have necessary provisions to support dynamic reconfiguration.

4.7 Logical Model of The RT-DOS Kernel

Among various models suggested for building a distributed system, the *Processor Pool* model [Wulf 1981] seems to be a natural choice for Transputer-based systems. The relatively low granularity of Transputer units, their connectivity, and high performance interprocessor communication facilities are among major arguments favoring this model. The *Processor Model*, also referred as *Processor Farm* [Helios 1989], assumes that whenever a service needs some computing power, one or more processing units are temporarily assigned to that service, then claimed back when the computation ends. The model supports strongly the idea of dynamic reconfiguration much demanded by adaptive systems.

Basic elements of the RT-DOS architecture (**Figure 4.7 (1)**) are those found in systems applying the Processor Pool model, i.e., Cambridge Distributed Computing System [Needham 1982], and AMOEBA [TanenbaumC 1981]. They consist of:

- Terminal Pool;
- Service Pool;
- Processor Pool; and
- Gateway(s).

Terminals may be plain VDUs attached to the system through special Transputer boards equipped with serial ports (RS232), or intelligent workstations (mono or multiprogrammed) equipped with Transputer link interfaces (for example, IMS B004 Development System).

The RT-DOS services (**Figure 4.7 (2)**) are distributed on various processing elements. Their location and number are dynamic and transparent to the RT-DOS users. Services are provided by:

- Terminal Servers, running user command interface(s);
- File Servers;
- Printer Servers;

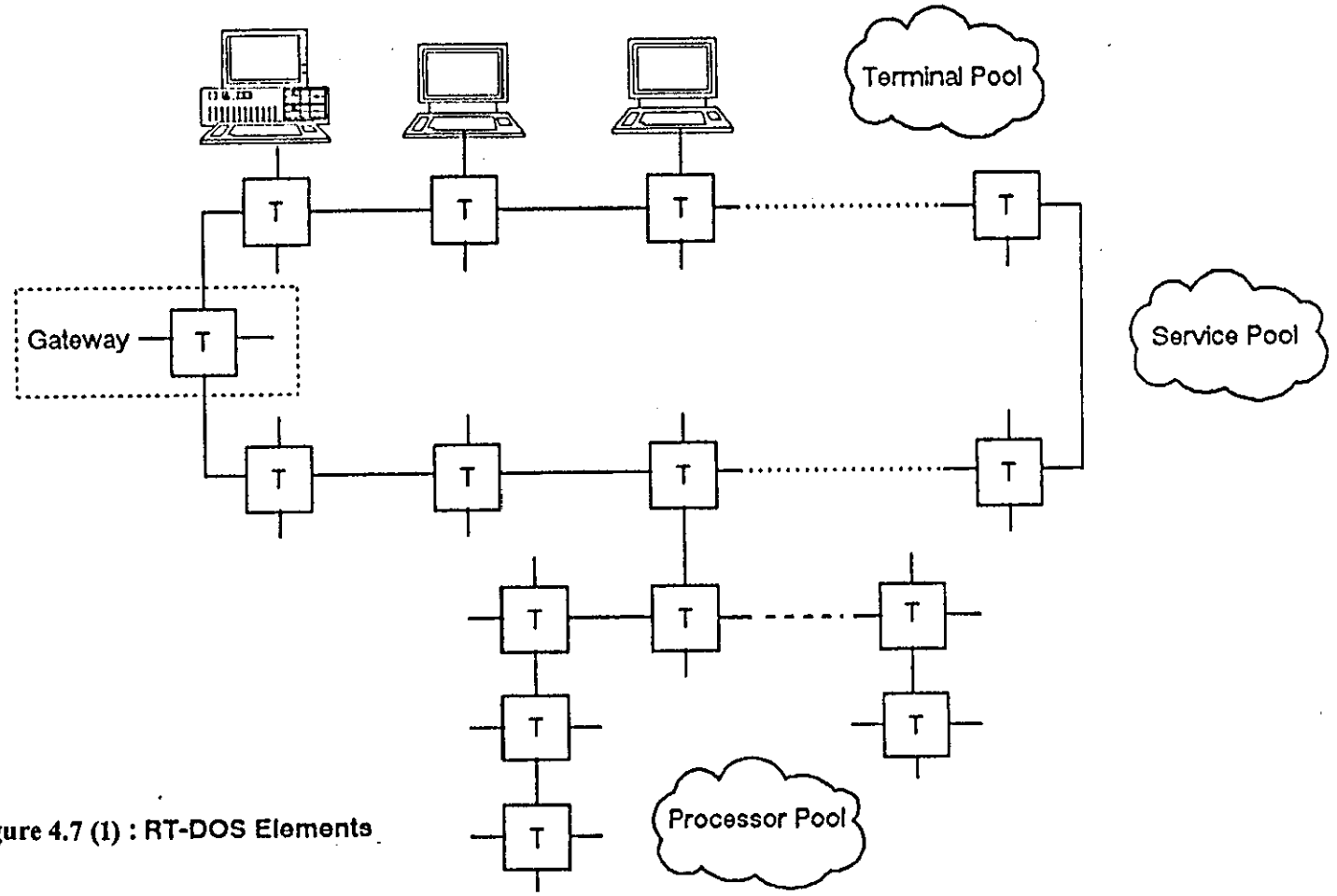


Figure 4.7 (1) : RT-DOS Elements

- Name Servers;
- Time Servers;
- Processors Pool Servers;
- Resource Manager(s);
- Network Configurer; and
- Switchboard (please refer to the physical model of the RT-DOS).

These initial services can be dynamically replicated and new services can be created by allocating additional processors from the pool as needed. The processor pool consists of 16/32 bit Transputers, optionally equipped with a floating point arithmetic unit. The pool is controlled by the Pool Manager running as an independent server.

The RT-DOS logical model envisions the implementation of real-time applications as separate clusters of processors. A given application claims the required number and type of processors from the pool, and forms an independent execution domain (please refer to **Figure 4.7(2)**). An *Application Manager*, running on a separate Transputer, is in charge of monitoring the domain and interfacing it with the service pool. The RT-DOS has a priori no limitation on the number of application domains. Their number can only be confined by global resource constraints, and other physical considerations. Moreover, a given application may consist of multiple domains, isolating for instance time-critical processes from non-critical and background tasks.

Gateways are optional extensions, connecting the remote RT-DOS sites which are beyond the reach of Transputer links (in excess of a few meters). They may consist of specialized Transputer boards, i.e., Ethernet interfaces, or intelligent workstations carrying out the message switching.

All processing elements in the RT-DOS will run the same kernel which will provide process, memory, time, interprocess communication, short-term scheduling, and statistical data exchange services. Each of these functions will be implemented by a separate kernel module. Kernels, replicated on each site, will cooperate to provide a network transparent single system abstraction.

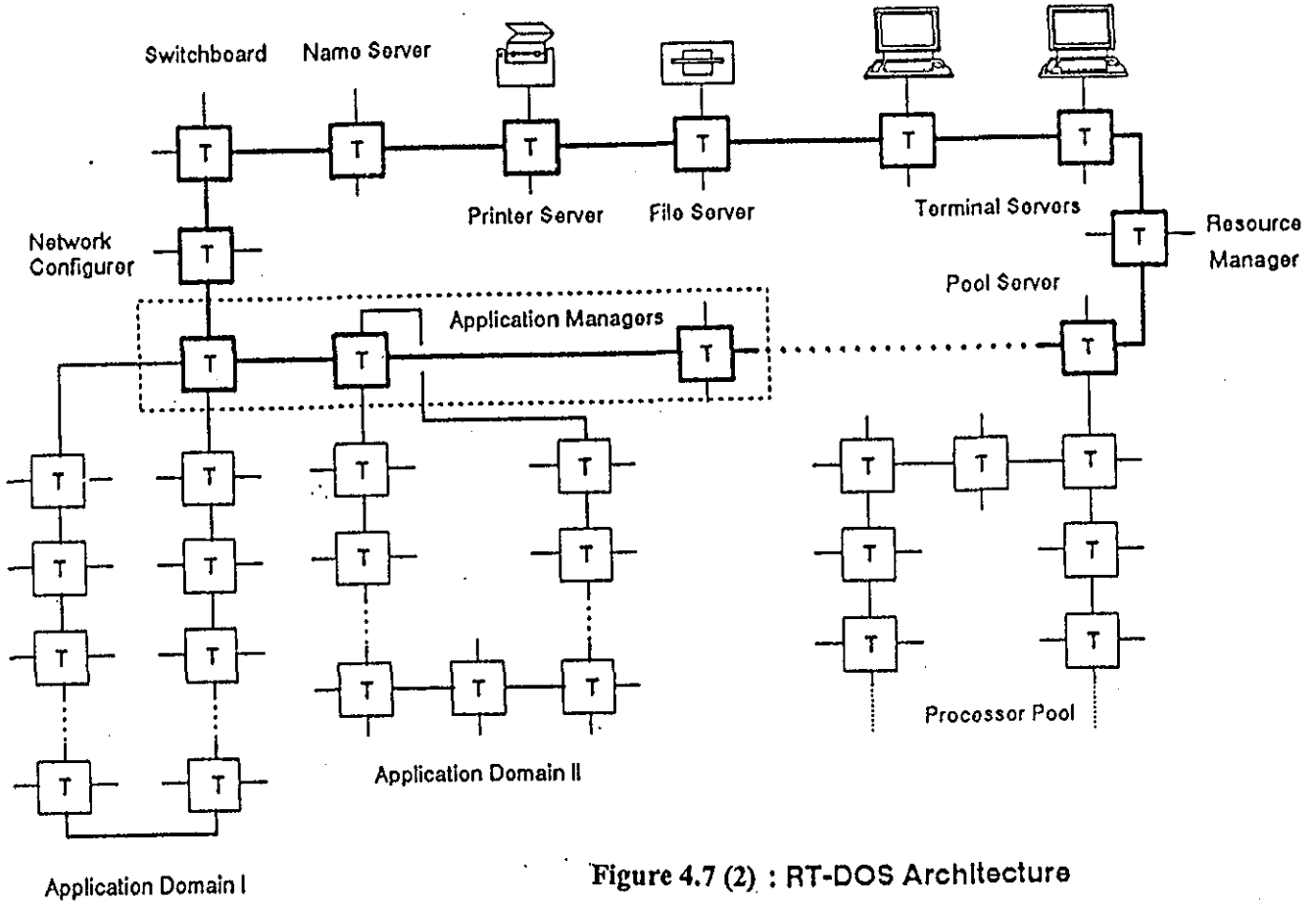


Figure 4.7 (2) : RT-DOS Architecture

RT-DOS services

4.8 Physical Model of The RT-DOS Kernel

The topology of the network, supporting the logical model defined in the previous section (**Section 4.7**), is a critical design issue. Transputers can be interconnected to form various structures ranging from pipeline, loop, array (2D, 3D,..) to the hyper cube. These structures vary in three important respects:

- Ability to be configured;
- Ability to be extended; and
- Performance of interprocess communication.

All of the above configurations are theoretically realizable on Transputer-based systems. Yet, the architectural restrictions of Transputer impose an upper physical limit on the number of connections at a given node as Transputers have a maximum of four physical links. Nevertheless, the topology of the RT-DOS network has to be designed considering firstly the high performance communication criteria, and then, extendibility and reconfiguration issues.

In a distributed system, flow of information among processes consists of control messages and data exchange. In general, control flow corresponds to short messages of broadcast/multicast type, whereas data flow requires lengthy transfers of point to point type. The RT-DOS topology takes into account the difference between these two category of messages. Control messages and commands are conveyed over permanent connections organized in rings, while processes exchange data via point to point connections established on request (please refer to **Figure 4.8 (1)**).

The physical model of the RT-DOS is based on several connected loops. The service loop forms the backbone of the RT-DOS architecture. Transputers, running operating system servers, are connected point to point via two of their four serial links. Application domains are organized as independent loops, connected to the service loop through the *Application Managers*. This multi-loop organization limits the number of messages in transit in a given domain and confines the length of communication paths. Control messages related to one domain, such as statistical and monitoring requests issued by the Application Manager, circulate within the domain with no effect on other parts.

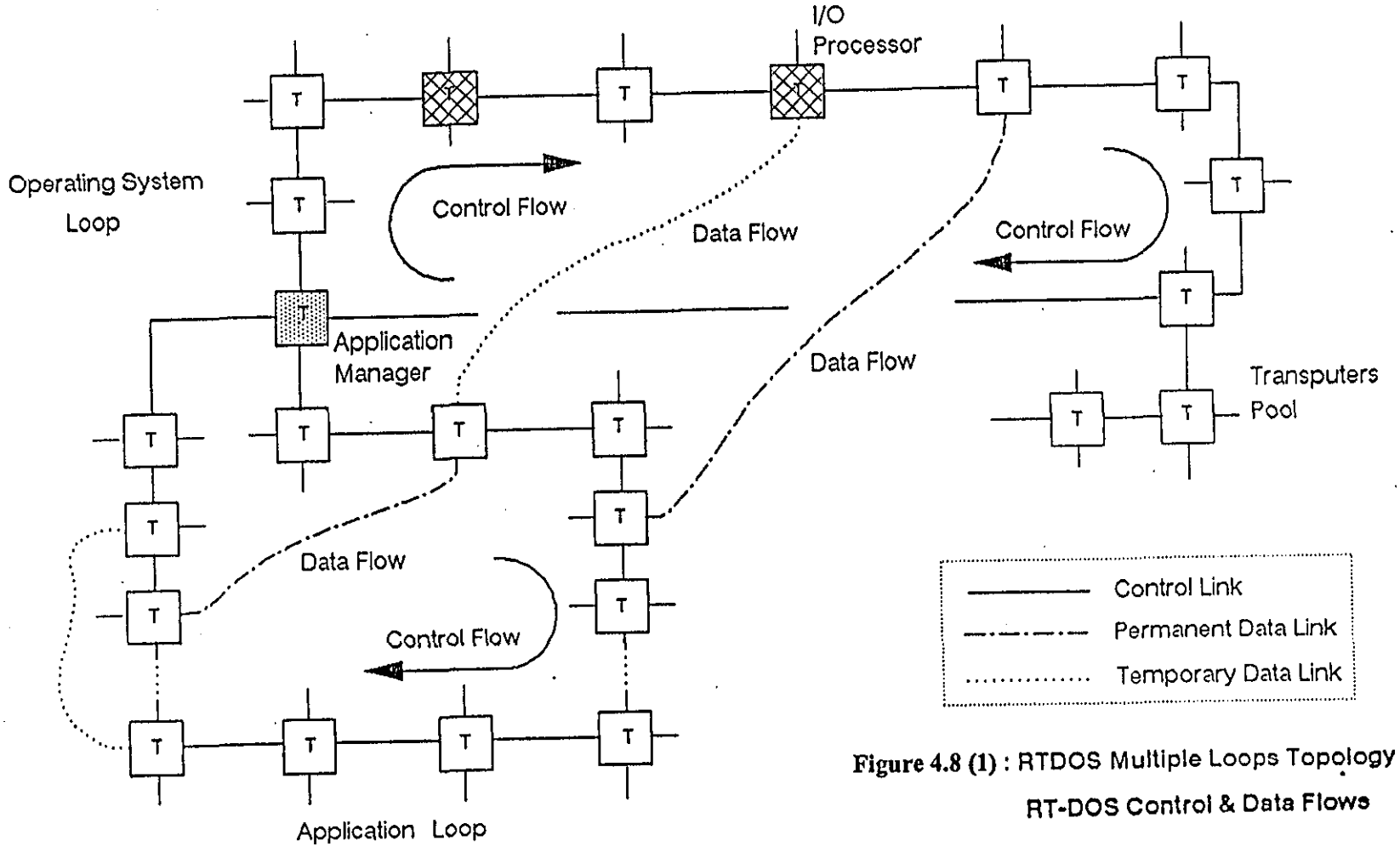


Figure 4.8 (1) : RTDOS Multiple Loops Topology

RT-DOS Control & Data Flows

Broadcast messages, issued for example by the *Name or Time Servers*, travel shorter paths to reach individual Transputers in respective loops, as it will be shown in the following sections.

Remaining Transputer links (two) are used in data transfers. The *Network Configurer* and the *Switchboard* services establish point to point communications on request, using programmable link switches. Thus, data from/to a file server for instance can be transferred to the concerned unit without disturbing other processing modules in the system. The model also foresees the allocation of permanent data paths, to carry time-critical or heavy data traffic.

The multiple loop architecture of the RT-DOS also contributes to the fault containment. Separate address spaces (confined in each Transputer memory), and private application domains enforce the protection and isolate to some extent the propagation of faults.

4.9 The RT-DOS Kernel Implementation Model

The RT-DOS Kernel is implemented using the server model as in the *V Kernel* [CheritonB 1983, CheritonA 1988]. Implementing kernel functions using a server model, replicating them in each kernel, and accessing them through the standard IPC interface has several advantages. These are:

- Implementation of each module is simplified, because each instance of the server module manages only local objects;
- A client can access the kernel servers the same way as other servers;
- The use of the IPC interface minimizes the additional kernel mechanisms for accessing remote kernel servers;
- The use of the IPC primitives to access these servers avoids adding some extra kernel traps beyond that required by the IPC interface. Besides avoiding a proliferation of system calls, this design simplifies the job of imposing and verifying the integrity and security requirements for the kernel;
- This design separates the IPC from other kernel services so that the IPC mechanism can be tuned independently of other less performance-critical services;

- The invocation mechanism is general in that additional kernel server modules can be added, as might be required in high performance real-time control systems.

The implementation model of the RT-DOS Interprocess Communication (IPC) mechanism is chosen to be a message-based system following the semantics of remote procedure calls (RPC). The IPC issue, as being the main subject of the thesis, is elaborated in the next chapter (**Chapter 5**), in detail.

5 DESIGN AND IMPLEMENTATION OF AN IPC MECHANISM FOR THE RT-DOS KERNEL

Interprocess Communication (IPC) is potentially the dominant component of performance in multiprocessor computer architectures. This has been responsible for developments in routing algorithms, interconnection techniques, and networking hardware equipment [Gaughan 1993]. High-bandwidth communication channels, such as optical fibers, offer Gbps transmission rates even between distant nodes located kilometers apart [Vetter 1993].

Design of an Interprocess Communication (IPC) model requires answers to the following questions:

- What is the communication entity (Port, Channel/Link, Mailbox) ?
- What should be the naming scope of communication entities ?
- How IPC naming schema should be related to general naming policy ?
- Should the IPC primitives be buffered or unbuffered, and where should be the place of buffering ?
- Should the IPC enforce error and flow control ?
- Should the IPC primitives be synchronous or asynchronous ?
- How are connections established between a process and the operating system, and between application processes ?
- Should the IPC be implemented following client/server or remote procedure call (RPC) model ?
- Should the IPC include provisions to support fault-tolerance mechanisms, i.e., process replication or migration ?
- What is the relevance of transmission type (broadcasting, multicasting, etc.) with respect to real-time applications ?
- Should messages be of fixed or varying length ?
- Should messages be typed (differentiating commands and data) ?
- Will data be typed in messages ?
- Will broadcasting and multicasting be supported or not ?

First we examine the question of the appropriateness of remote procedure call or client-server architectures, and then elaborate on the other issues in the following sections. It is to be noted here that the message-based communication can be implemented either following the semantics of remote procedure calls (RPC), or applying client/server

model. It is also very well known that asynchronous primitives impose buffering, and client-server systems are built mostly using asynchronous communication protocols.

In client-server models the request is queued for processing should the addressed server process be busy when the request is received. This model appears to be preferable when serialization of request handling is required.

The remote procedure call model is similar to procedure calls in programming languages. Each request can be visualized as the transfer of flow of control from the caller to the invoked procedure. The idea is to make the semantics of interprocess communication closer to well understood procedure calls. The procedure invocation is preferred when there are significant performance benefits to concurrent request handling. However, the RPC is a semantic extension of message-based communication model. It can be implemented on a system applying client/server communication model.

In the following chapters, design and implementation of a predictable IPC mechanism will be presented.

5.1 Design Considerations of the RT-DOS IPC Mechanism

The backbone of any Distributed Operating System is its message passing facility referred to as IPC (Interprocess Communication). For non real-time distributed operating systems, the IPC facility is assumed to support the correct and eventual delivery of messages to given destinations with no consideration of deadlines. For real-time distributed operating systems, not only correctness but also timeliness of message delivery are the key design issue for an IPC facility [StankovicB 1989, HideyukiB 1987]. It is generally recognized that, in real-time environments, it is preferable not to deliver a message at all than to deliver it late. Timeliness of message delivery can be defined as the correct delivery of messages to application processes within application dependent prespecified delays. For static real-time applications with stable task populations, simulation and exhaustive testing are traditionally used to measure the timeliness of message delivery (and task scheduling as well); changes are made where needed to meet application deadlines. If predictability of message passing is defined as the possibility of determining, with certainty, the arrival time of a message after it is

sent, then it becomes quite clear that what is needed in dynamic environments is a predictable message passing facility.

The thesis work is based on an investigation of the feasibility of providing a predictable message passing facility for a network of Transputers with a predefined topology on which a predictable distributed real-time operating system (RT-DOS) kernel can be built. The targeted IPC mechanisms were sought to support predictability, fault tolerance, location independence, and yet high communication bandwidth. A micro kernel replicated on each node of the Transputer network will include the IPC mechanism as its message passing service between all upper processes. Anything else, except the micro kernel will be considered as a user process, including upper layer operating system policy services. Application domains and the system domain will be isolated from each other to eliminate communication bottlenecks, through domain loops. Between application domains and/or operating system services, communication will be carried out through permanent links, unreliable datagram services and temporary communication link channels established dynamically on demand between communication entities on top of the datagram service.

Dynamic short-lived stateless circuits established on need between a dynamically changing task population, will support building of dynamic systems, provide fault tolerance, and flexibility in link allocation. Mapping of logical circuits to physical Transputer links will be managed by a centralized switchboard system. Logical circuits will be established between processes to provide high bandwidth, while the datagram service will be used for exchange of control messages between kernels and to support group communication (such as broadcasting and multicasting) between communication entities.

Finally, the IPC mechanism provides unblocked and blocked RPC calls to support non-predictable communications. All reliability issues are assumed to be handled at process level, by end-to-end control protocols, and are implemented according to the need of each process type. Long term message buffers are not maintained to keep the kernel in a minimum size. To increase reliability and adaptability, and to reduce the reconfiguration and error recovery costs, no communication state information about connections or links is maintained long term. The address of none of the communication entities is made known to any of the processes to provide location transparency and topology

independence. Only a name server has a publicly known name (but not address), and all other communication entities and processes locate each other through the name server which finds locations using a message broadcasting protocol based on an unreliable datagram service.

It is assumed that the congestion problem, system wide global timing service, and other system housekeeping and statistics services are handled by the upper kernel policy layers implemented as user processes. The IPC mechanism provides these upper layers with message transport service only, regardless of the end-to-end protocols which are used by them.

5.2 Elaboration of the Existing IPC Design Approaches

Though most of the existing approaches of DOS kernels to the IPC design, such as V (VMTP) [CheritonC 1986], Helios [Helios 1989, Grimsdale 1989], Accent [RashidA 1981], Alpha [Nortcutt 1987], ARTS [HideyukiB 1987], StarOs [Jones 1979], Mach [Accetta 1986], Medusa [Ousterhout 1980, Ousterhout 1980], HYDRA/C [Wulf 1981, HYDRA/OS 1975], iMAX [Khan 1981], Wisdom [Murray 1988], and Charlotte [ArtsyA 1987], have been studied in detail, the three most relevant (VMTP, Helios, and Wisdom) are summarized below to give an idea about the feasibility and/or drawbacks of different IPC design approaches.

VMTP is a transport-level protocol designed to support remote procedure call (RPC), real-time datagrams and multicast communication, for the V kernel. The *VMTP* is basically a request-response protocol. A *VMTP* session, a message transaction, is initiated by a client sending a request message to a server entity and terminated by the sending back a response message. The response acknowledges to the client receipt of the request message. The next request from the client, explicit acknowledgment or time-out acknowledges to the server receipt of the response by the client. A client can only have one message transaction outstanding at one time although a host may implement multiple *VMTP* clients. Request and response may consist of multiple packets.

A *VMTP* message transaction takes place between-visible entities, client entities and server entities. An entity may be a process, port or procedure invocation. Each entity is addressed by a 64-bit entity identifier. A group of entities (a group of file servers) can be

identified by a single entity identifier even if they are distributed across several machines.

VMTP provides facilities for the higher level modules to implement conversations directly. The two key aspects to *VMTP* conversation support are : stable addressing and message transactions. A stable address is one that retains the same meaning or binding as long as it remains valid. A message transaction is a request-response pair with reliable delivery on both the request and the response message. *VMTP* is defined in terms of a datagram delivery model of a network or inter-network. Conversation support in *VMTP* is preferred over directly implementing connections, as in TCP, for several reasons:

- Minimal redundancy;
- Minimal server state;
- Minimal client state;
- Flexible higher-level conversation.

VMTP assumes an underlying delivery service that provides end-to-end (best-effort) datagram delivery, such as provided by IP and raw Ethernet. It is also designed to take advantage of datagram multicast facilities such as available on the Ethernet. The packet layout is logically structured as 4 portions:

- Entity and transaction identification - including authentication identifier, domain, source, destination, forwarder and transaction identifier;
- Packet group control - including checksum, control flags, segment offset, function code and delivery mask;
- User message control block - system flags, user data and segment size;
- Segment data - a maximum of 16 kilobytes of segment data, possibly further limited by the maximum packet size.

The basic *VMTP* header (without data segment) is 76 bytes. Including segment data, a *VMTP* message can be as large as 76 bytes plus 16 kilobytes for segment data. However, the actual transmission size includes the IP header, network header plus the 76-byte header replicated in each packet. *VMTP* is currently being reviewed as a candidate for the Internet request-response protocol by the Internet End-to-end Task Force.

Helios is a multiprocessor, multi-user distributed operating system. It efficiently controls the resources of multiprocessor architectures and provides an advanced interface to multiple users of such architectures [Grimsdale 1989]. The system model which Helios sanctions lies somewhere between the workstation and processor pool model of Tanenbaum. Networks may be composed of loosely connected Transputer workstations, which may contain one or more Transputers. Although the principles are applicable to any distributed memory architecture, the *Helios* implementation is the first example optimized for the Transputer. It provides a dynamic processor allocation strategy. It consists of a small message (fixed length) passing nucleus located on every processor, onto which are added other high-level system services. It provides a robust, but essentially simple, basic transport layer on top of which can be built more complicated protocols and applications.

The *Helios* kernel consists of a number of separate parallel processes and supporting system procedures. There are four link guardians, one allocated to each Transputer link. The link guardian is responsible for processing incoming messages. It first reads the message header and then, depending on the destination port, it delivers the control and data vectors to the relevant destination, which may be a destination within the current processor or a physical link, in which case the message is transferred to a neighbor. The kernel also contains procedures for memory allocation, semaphore handling and event handling.

The *Helios* has system libraries (both resident and shareable) which provide a general interface to the operating system with the following system calls:

Open Opens a stream to a named object;

Locate Locates a named object and generates an object structure for subsequent management of that object;

Create Creates an object of a given type;

Link Establishes a link to a named object (symbolic link);

Delete Deletes an object;

Read Reads data from a stream into a buffer;

Write Writes data from a buffer onto a stream;

Seek Moves the file pointer.

Helios has a minimum of four I/O controllers (IOC) in each processor nucleus which act as I/O agents for tasks in remote processors and manage in-coming search requests on the four links. Each nucleus contains a hierarchical name table which is used to map ASCII network names onto port descriptors, thereby making it possible to access the local port table. Every item in a *Helios* network, whether it be a processor, a file system, a disc partition, a file, a task or an application program, uses the message passing facilities of the nucleus and are ultimately managed by software components of the nucleus. The message passing system (a simple send-receive protocol) provides unreliable, asynchronous, blocking (explained below), point-to-point communication. No automatic retransmission of messages is attempted by the system if a message is lost; the protocol must therefore be assumed to be unreliable. Because the message may pass through several intermediary processors, and because there is no end-to-end acknowledgment, the two communicating processes may not necessarily synchronize during the message transaction. Communication within *Helios* is essentially asynchronous: if the sending process makes a *send* request, it is blocked on a Transputer channel until a remote process is ready to receive on this channel; if the channel is a Transputer link, the sender will block until the kernel on a neighboring processor is ready to receive the message; and if the channel is local, the sender will block until the client is ready to receive the message. Processes are similarly blocked on *receive*. The sending process does not necessarily block until the receiver has received the message, but merely until the message has begun its journey to the receiver. Helios does not support a *broadcast* mode; every message must be provided with a single destination, so that all communication is point to point.

Helios kernel message passing primitives are called *PutMsg* and *GetMsg*. All messages consist of a fixed-size header, followed by an optional variable-size control vector and data vector. The control vector is used to transfer control information, e.g. file pointers, file size and object status; the data vector contains all the message data. Ports are allocated dynamically to user processes as required, and are maintained in a port table, which expands and contracts dynamically as ports are allocated and released. Each kernel manages its own local port table, and this table contains entries for objects within the processor or within one of its immediate neighbors. The port table is essentially a local routing table, which together with every other kernel port table forms part of a large dynamic, distributed routing table. Each message header contains information regarding the message destination, the size of control and data vectors, and the message

function. As a port supports only unidirectional communication, when a bi-directional stream is to be established each surrogate port on the transmit path must be mirrored by a reply port.

Two significant features of the *Helios* kernel IPC design include the omission of a reliable data link layer, and the provision for variable-size packets at the physical layer. In a *Helios* network the processing units are further subdivided into a number of subnetworks in a hierarchical manner, and wherever possible closely connected processors are located in the same subnetwork. Requests for objects which lie outside the local processor result in a distributed search that extends into the network. This search is implemented using a flood search technique, whereby individual search requests are issued from all four Transputer links to the link IOCs on neighboring processors. Searches which reach the edge of the network without finding the relevant object are terminated and a failure acknowledge is returned. The facility exists for any program to register itself as a system service. This is supported by a well defined general server protocol based on the concepts of the client server model. Servers are designed to be stateless, with all unrecoverable states maintained by client processes. The capability mechanism is used by all system servers to authenticate incoming requests and to ensure that a client can not exceed its original rights of access.

Because of the space limitations reserved for discussions of the existing IPC design approaches, the *Wisdom* [Murray 1988] is explained as the last example. It is a distributed operating system proposal for a network of Transputers implemented in a proposed version of OCCAM language. In *Wisdom,* a module is a collection, or cluster, of small, lightweight processes that cooperate to provide the services attributed to the module. The processes that make up a module are OCCAM processes, and the module is a single OCCAM program. When reference is made to interprocess communication, what is meant is inter-module communication, and not, communication between OCCAM processes via channels.

In *Wisdom* the software that comes closest to being the kernel is the combination of the three modules: routing, naming, and load balancing. Of these three the router is the only vital component, the rest could fail or be removed and the computer could still function. This is a logical extension of the trend to remove from the kernel non-vital services.

The OCCAM channel is the base of the *Wisdom*'s IPC, but as communication between modules may involve communication between many OCCAM processes, the end to end communication is not necessarily blocking. No replies are inherent, nor is delivery guaranteed: if these are required it is expected that these will be built on top of the system provided. From this base any other form is built by adding modules before and after the message enters the IPC support (the routing module). The three modules (Router, Naming, Load Balancing) of which the *Wisdom* is built provide an abstraction of the machine that user level software need not be aware of the hardware topology.

The purpose of the *Router* (routing module) is to allow any processing node to talk to any other. This is the only module that must appear at every node, and it does not require the services of any other modules for it to operate. It is different from routers in wide area networks. Given that the nodes in a Transputer network are also the hosts that perform the work of the system, it designed to keep the overheads involved in communication to a minimum.

The naming module (*Naming*) provides the interface between the router and other processes. It generates a connection between a server and a client, using the *Router* as necessary. The *Naming* module is built above the router (i.e. the Router is required by the naming module for it to operate), and makes use of the changes to OCCAM proposed there.

Load Balancing module attempts to give an optimum performance by sharing the workload as evenly as possible between the processors available, aiming at supporting hardware topology independence.

The *Wisdom* routing algorithm doesn't use a routing table because of the following considerations (in comparison with local or wide area networks):

- Error rates in message transmission are far lower;
- The topology is not free (in the current version the topology is fixed in structure, not size);
- It is unacceptable to have routing information at every node;
- Processing costs of routing must be low.

The Wisdom has an efficient deadlock and starvation avoidance algorithm. When all the buffers in a node become full, the node doesn't accept further messages unless the message has a shorter distance to travel than one of the those in its buffers. In this case the message with the greatest distance to travel is preempted. When a message arrives at its destination it is consumed. This requirement can be met by placing a time-out on all messages arriving at their final destination. If the message is not consumed during the time-out period it is discarded. It is possible to give each message a priority which will adjust its likelihood of delivery, and so reduce or eliminate the possibility of starvation.

5.3 Basic IPC Components

The RT-DOS Kernel and its IPC mechanism has been based on message passing technique, mostly because of its hardware structure and network topology. Problems related to the message passing systems remain the same regardless of the implementation specifics. Therefore, it is found useful to present below, a brief summary of the message passing systems and their features, as an introduction.

Issues of determining the semantics of message passing are:

- Process naming;
- Message sending policy (blocking versus non blocking send);
- Message representation (formats); and
- Handling of communication failures (error recovery).

The semantics of message-passing can be summarized as following:

a) Identification (naming) : for the sender objects and receiver objects, such as process, port, channel, and mailbox. This determines the search algorithms of the IPC;

b) Structure of messages : size, typing (typed, untyped), segregation between message and data;

c) Message distribution mechanisms : sending/receiving messages via intermediaries, search algorithms, the protocols between sender and receiver parties, and message primitives;

d) Error handling : recovering from lost messages, handling duplicated messages and network failures.

Identification: Identification issues are handled at two different levels: symbolic (user/process level), and absolute (low/IPC level) identifications (names) of communicating objects. The sender need not be aware of the absolute identifier of the recipient. A mechanism for mapping from symbolic to absolute identifier may be implemented at the sender's host node by use of name tables or name servers.

There are three basic categories of identification: name, address, and route. Name refers to an objects symbolic identification; address refer to absolute name of the object, namely its location in the system; and finally, route provides the means of finding the address (location) of the object. The Route may be entered in the identifier or implemented as a separate strategy, as in the RT-DOS IPC implementation.

Universality of identification schema is also an important issue. A unified global naming schema reduces the burden on the message handlers, while a distributed naming schema will place greater responsibility on the routing algorithm.

Structure of messages: Message size may be fixed or variable; and messages may be structured into headers, control, and data, or may be combination of these. The protocol to be followed in message exchange may either be rigidly fixed by the IPC mechanisms, or left to the end-user processes (end-to-end protocols, as in the RT-DOS IPC). Message typing may or may not be emphasized, depending on the targeted network's type being heterogeneous or homogenous.

Decisions taken on these regards are based on the consideration of the IPC size and its size in terms of functionality. The trend lately is that, the IPC mechanism should be minimal with a very well defined functionality, and decoupled from any policy related services.

Message distribution mechanisms : Message identification and message structure have little influence on each other, but they have a direct bearing on the message-passing mechanism. The message passing mechanism has two components: nodal mechanism (intra-nodal), and routing (inter-nodal) mechanism.

Two most important aspects of the nodal mechanism can be identified as: *mode of communication*, and *synchronicity of message passing*. Mode of communication can be

direct (producers are themselves dealers) or indirect. Indirect mode can be implemented through ports, queues, and channels. Indirect communication allows greater flexibility and robustness of operating system services, because of the extra abstraction layer between communication service and the process. Naturally, this extra abstraction layer introduces a performance decrease in the communication services. This is not a surprise because the flexibility and performance conflicts in general.

Synchronicity of communication can be blocking or non blocking. Synchronicity may be dictated at the lowest level by processor (e.g., Transputer link communication), but asynchronous communication can be built on top of synchronous communication, as in the RT-DOS IPC. Synchronous mechanisms are easy to implement and provide more precise predictability, but make it very costly to implement group communication, such as broadcasting or multicasting.

Naming forms a very important component of the IPC mechanism, and careful routing increases efficiency of inter-node communication time, especially for big networks. The overhead of routing affects all the intermediate nodes and the processes executing on them, even if they are unrelated to the message.

Routing consists of four components:

- Search algorithm (relevant to naming);
- Reliability;
- Relay versus store and forward of messages; and
- Datagram versus virtual circuit transmission.

The search algorithm is dictated by the name/address/route schema of the naming service, and the topology of the computing resources. The relay approach implies that message forwarding begins immediately after the address fields of the message are received. Virtual circuit transmission is not possible unless the path is known prior to sending the message. However, once a message has been sent to a recipient, the path is known, and may be stored for later use (at least for a T-time period) to increase efficiency.

Routing strategy is strongly influenced by the object identification policy, and efficiency of routing is the most important issue in a distributed operating system IPC implementation, as it is the backbone of the whole system-wide interacting services.

If names of the objects provide the path between the sender and the recipient, then no search algorithm is needed; message routing reduces to the simple mechanics of physical transmission of the message between successive pairs of intermediate nodes. However, this might increase the message size, cost extra data transmission, and decrease the flexibility of migration of processes between nodes. If the name gives the address of the recipient but no path, then an additional layer of software is required to implement the search algorithm on each node. This may cause a memory buffer problem for nodes.

In most DOS implementations, naming at the process level is symbolic, giving neither address nor path (mapping from symbolic name to network-wide identification requires a software device, such as name table, in every resident kernel).

Error handling : Ascertaining the reliability of message reception and correcting errors is often left to the end-users (upper services). They use time-out techniques to implement elementary control over reliability. If the underlying network reliability is very high, then implementing reliability checks in higher levels will increase the efficiency of transport service, by reducing the inter-node computation costs. For unreliable network medias, retransmission of bigger messages on upper levels is not acceptable. Message serialization will also affect the cost of transmission, as it will mostly depend on a centralized serialization service which is one of the bottlenecks of distributed operating systems.

Ports seem to provide suitable abstractions to represent the IPC entities, and ACCENT [RashidB 1981] and AMOEBA [TanenbaumC 1981] offer good examples of this approach. They provide a global and uniform interface to system objects and services. Process identities become transparent and services provided can be readily transferred from one server to other.

UNIX based systems, such as LOCUS [Walker 1983], extended the existing IPC and file management primitives to support network wide communications. However, in many

cases this approach seems to be non-flexible and restrictive, as it inherits the limitations of the existing system.

5.3.2 Naming Schema of IPC Entities

An IPC mechanism has to have identifiers (addresses) for all processes in order to perform its message delivery function. An address is bound to an object by routing context. Other common terms used for addresses are ports, sockets, and mail boxes. It is convenient for a process to have more than one communication address, allowing parallel communications with other services, or partitioning of services (multi purpose functional processes). It is also useful to associate one identifier with several processes so that services can be replicated or relocated.

In case port identifiers contain location dependent information, process migration causes messages to be sent to obsolete addresses. As port identifiers could be disseminated all over the network, it is not feasible to try to update them during the migration all at once. One solution is that the source site continues to forward the new location (if it is still able to do so) after a process is moved, as in DEMOS/MP [Powell 1983]. As messages are forwarded, the old kernel can notify the originator's kernel of the address change and avoid subsequent rerouting process. The forwarding address is needed as long as old references still exist, or the migrated process is alive. Its removal can be an event based process or a garbage collection activity. An alternative to message forwarding is to notify the sender of the absence of an addressed process. The sending kernel can attempt to find the new location through process managers or a system wide name service. The main disadvantage of the approach is that more of the system is involved in message forwarding, and would be aware of process migration.

Name binding, or initial connection establishment, between communication entities is another important issue. There are two categories of connection to be considered:

- Between a process and the operating system (kernel); and
- Between application processes.

Communicating processes can establish a connection either through the operating system's name servers (public name services or private arrangements), or inherit them from parent processes. As for process to operating system kernel communication, the

majority of systems adopt the static binding schema which establishes connections (umbilical ports) at the creation of the process.

5.3.3 Communication Primitives and Protocols

Determining the semantics of communication primitives is a controversial issue. Fundamental design decisions relate to:

- Reliable versus unreliable primitives;
- Blocking versus non-blocking primitives; and
- Buffered versus unbuffered communication.

With unreliable communication primitives no guarantee of delivery is provided, and no automatic retransmission is attempted if the message transmission or replication fails. On the other hand, the communication subsystem may try to provide error free communication channels. Although the error-controlled, flow-controlled *Virtual Circuit* type communication model seems to be attractive, there are strong arguments in the favor of uncontrolled, connectionless datagram-like services. A major tenet in this direction is that, since high reliability can only be achieved by end-to-end acknowledgments at the highest level of protocols, the lower levels need not be 100 percent reliable. Moreover, the experience with RIG [Ball 1976] has shown that the exact flow control policy is much less important to overall system performance than the higher level communication protocols devised to solve individual problems.

Blocking (synchronous) communication primitives return control when the message is transmitted or received or an error condition is detected. Non-blocking (asynchronous) primitives initiate the request and return the control to the requester. An asynchronous mechanism signals the completion or occurrence of the event or error condition (transfer completed, message arrival, time-out, etc.). Although non-blocking primitives offer maximum flexibility and overlapped actions, the blocking version seems to be a reasonable choice for several reasons:

- They carry the same semantics as procedure call, therefore easier to use;
- They simplify the buffering because data is kept in client's buffer, and the answer can be delivered directly into this buffer;
- They simplify transport-level protocol, because both error handling and flow control exploit the response to acknowledge a request and authorize a new one.

Unbuffered communication forces primitives to be synchronous. The sender is has to wait until the receiver retrieves the information (rendezvous). However, unbuffered communication can also be implemented as asynchronous primitives with due system support by restricting the maximum buffer size created run time [TanenbaumD 1985]. Buffered communication increases the concurrency level and message throughput. However, the existence of buffered information is often a handicap for fault tolerance and process migration. It is also more complicated to implement, because of error recovery and other system maintenance issues (synchronization, serialization, congestion, buffer management, etc.).

5.3.4 Message Structures

In a direct network architecture (as is the case for the RT-DOS) nodes do not physically share memory, they must communicate by passing messages through the network [Ni 1993]. Message size may vary, depending on the application. For efficient and fair use of network resources, a message is often divided into packets prior to transmission. A packet is the smallest unit of communication that contains routing and sequencing information; this information is usually carried in the packet header.

Naturally, there is always a fixed overhead with preparing, sending and receiving messages. Therefore, long messages reduce the ratio of fixed overhead, but also decrease the availability of communication media, much needed for control-type messages and interactive applications. A trade off between two extreme approaches (short size messages versus long ones) should be obtained considering the requirements of the application services which will use the communication mechanism.

The distinction between small messages and a separate data transfer facility ties well with a frequently observed usage pattern. The majority of the IPC activities are related to the transferring of small control/information messages between distant kernels and/or processes, while occasionally there is bulk data transfer (e.g., program loading, process migration, etc.).

It is known that small fixed size messages reduce queuing and buffering problems in the kernel. Ideally, only small and fixed size message buffers should be allocated in the

kernel and large amount of data are transferred directly between processes' address spaces, without extra copies (without copying the message through intermediate nodes).

Finally, data typing is needed to pass system objects implicitly in messages and to provide automatic data conversion in heterogeneous systems.

5.4 Implementation of the RT-DOS IPC Mechanism

The RT-DOS IPC services, being message based services, are built on message switching and circuit switching mechanisms that consider and utilize the multi loop topology of the underlying network transmission media. The objective was to obtain in the order of millisecond level (considering performance of the current state-of-art Transputer hardware) connection and delivery times of messages, between physically distributed processes anywhere on the network, regardless of its size.

From the very beginning, the predictability of the overall system was the prime concern of the design policy (as it should be expected from any real time system) for the RT-DOS kernel. During the implementation of some other factors, which can affect the predictability of communication negatively (though they increasing the flexibility of the upper layers and satisfy quite different system requirements), proved to be important to be considered at the very early stages of the design of the IPC. These are group communication support mechanisms, such as broadcasting, multicasting, and domain casting; and non-reliable message distribution mechanisms which require no replies from message recipients. Though at the very beginning it was accepted that predictability of the IPC services could be guaranteed only by setting point-to-point connections (circuit switches) between processes and transferring messages directly from one process space to another process space, later it was discovered that the number of messages exchanged between communicating parties to establish a direct connection was much more costly than sending the message itself through the network to the destination, especially for small messages that are less than 2KB, as it is shown in the following chapters (**Chapter 7**).

In the following sub-sections all these implementation factors are discussed in detail, and the basic IPC components (protocol layers, message structures, communication protocols

between the basic IPC entities) with the underlying data structures, required to support them, are presented.

5.4.1 Communication Model

The RT-DOS has been designed to run on Transputer systems configured as a collection of interconnected domains (**Figure 5.4.1(1)**). An RT-DOS domain is formed by connecting two out of four of the communication links of a Transputer to adjacent nodes. The resulting communication loop, referred to as the *Control Loop* or *System Loop* (or *OS Loop*), is used to convey interprocess communication traffic between kernel processes.

Though system services can be located anywhere (on any node or more than one nodes) most of them are expected to be on the nodes of the *System Loop* to isolate them from the unnecessary traffic of application domains which are self-contained.

Application domains are formed dynamically and attached to the *System Loop* as secondary level communication loops. All application domains share the same system services located on the *System Loop*, while most of the communication between the processes of the same domain are retained in the domain loop to guarantee a certain level of stability in terms of functionality of the global system services.

The System Loop is not used only for exchange of control messages between kernels, but can be used for the transmission of messages between processes which are physically located on different domains, especially for short length messages. Datagrams related to message broadcasting or multicasting, should also travel through the *System Loop* to reach their destinations.

Each node (Transputer) on the *System Loop* or any *Domain Loop* is connected to the previous and the next nodes via two of its four serial links. The two remaining links of each node can be used to dynamically connect to another node's links for point-to-point data transfer between processes on different nodes (on the same or different domain), or can be used permanently to connect them to outside hardware devices, such as terminals, disks, sensors, actuators, and other Transputers (for efficiency).

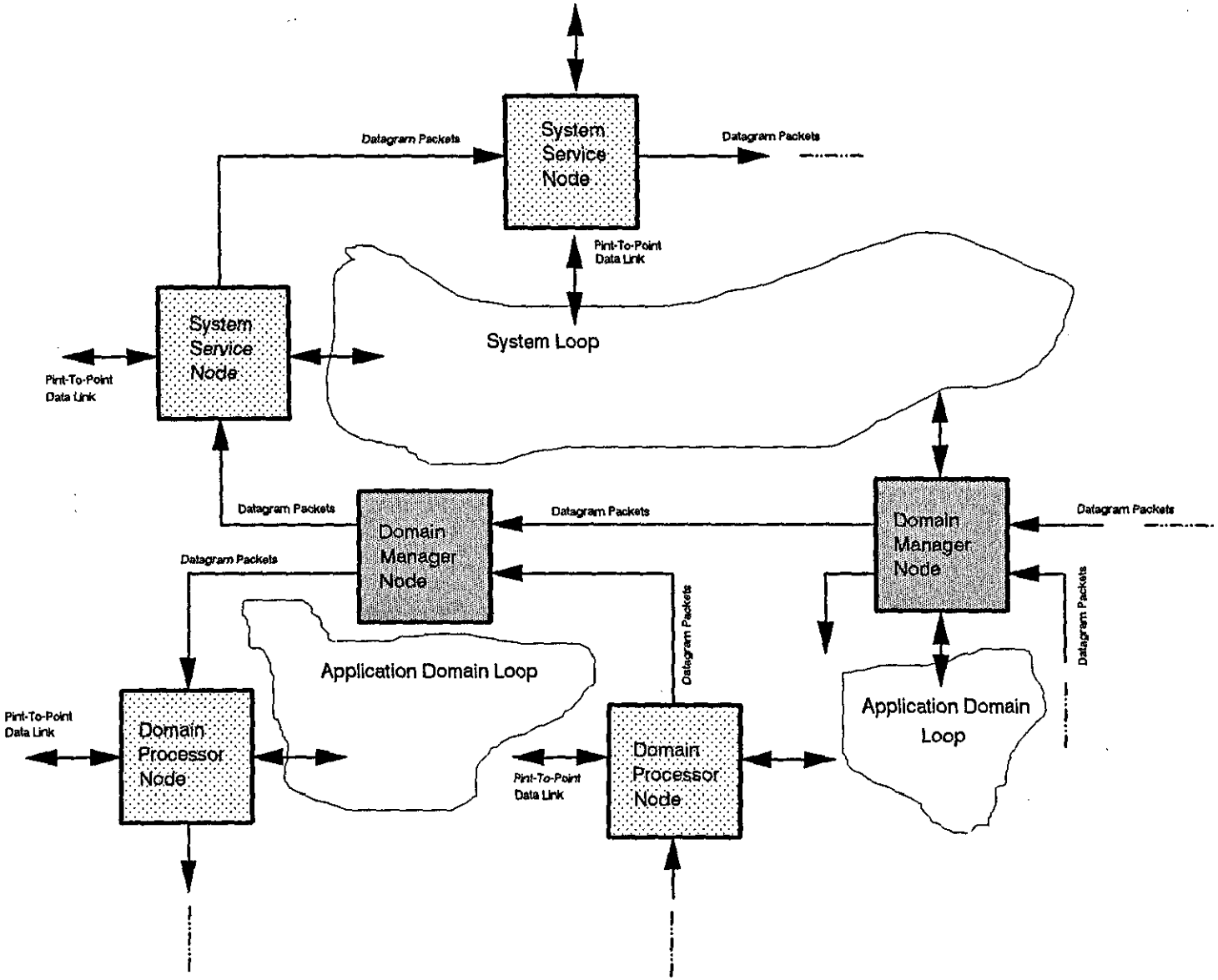


Figure 5.4.1(1) : RT-DOS Multi-Loop Topology and Connectivity of Physical Links

The nodes which are both on the *System Loop* and any of the *Domain Loops* are dedicated to the management of the related application *Domain Loop*, and are gateways between the two intersecting loops. These nodes are called *Domain Manager* nodes and they don't have any extra link to connect them to the outside events. Functionality of the IPC mechanisms on these nodes are different than the others, in terms of message routing policy only, though the same copy of the C code which runs on every node also runs on them.

As the full functionality of the RT-DOS Kernel is the same on all nodes (replicated kernel code), any node can be assigned as Domain Manager at any time dynamically, by the related upper level RT-DOS services. By switching a status bit (*NodeType*) on or off, a node can function as an ordinary one or a *Domain Manager*.

The RT-DOS architecture does not specify functional attributes of individual domains, nor does it imposes a priori limits on either the number of Transputers in a domain, or the number of domains in a system.

The most important system services related to the IPC implementation, residing on the *System Loop*, are the *Connection Server* and the *Name Server*. These two are directly involved in the connection establishment protocol of point-to-point process communication. The *Name Server* implements a more general purpose global system service and is not a direct part of the IPC mechanism. The *Connection Service* is used only to implement the circuit switching service, and is one of the major components of the IPC mechanism. All the other IPC connectionless communication protocols, as well as the connection oriented circuit switching protocol itself, are based on a simple datagram service.

5.4.2 Topology of the Underlying Network

In a direct network architecture, each node has a point-to-point, or direct, connection to some number of other nodes, called neighboring nodes. Direct networks have become a popular architecture for constructing massively parallel computers because they scale well; that is, as the number of nodes in the system increases, the total communication bandwidth, memory bandwidth, and processing capability of the system also increases proportionally [Ni 1993].

The multiloop topology of the RT-DOS derived, to a great extent, from the concern to provide a scaleable architecture that supports the effective and predictable IPC services. As in any other direct network architecture the overall processing power, memory capacity, and communication bandwidth of the RT-DOS systems grow proportionally with the added elements. On the negative side, the performance of communication over the *System Loop* is hindered by the size of a given loop, since network latency introduced by this topology is proportional to the distance between the source and destination nodes. However, the RT-DOS architecture limits the impact of serial propagation delays, as the average distance between nodes can be kept significantly low by partitioning the system into multiple domains.

In message-based systems, predictability (which is a vital requirement of real time systems) of communication services is negatively affected by the frequency of exchanged messages and the amount of transmitted data. The RT-DOS architecture aims at minimizing the impact of these unpredictability factors by assigning dedicated communication paths to control and data flows and by confining the local traffic within domain boundaries.

The provision of direct communication channels over switched links allows the use of native Transputer I/O primitives between processes distributed over the network, with immediate implications such as availability of high bandwidth communication channels (vital especially for bulk amount of data transfer between processes, such as file transfer), simplicity and efficiency of system design and total network transparency. Moreover, the ability to transfer data directly between source and target address spaces eliminates the involvement of intermediary elements, such as network routers and local kernel processes. Bypassed system elements are relieved from the heavy communication load. Also unpredictability introduced by network congestion and message switching overheads are avoided.

Unfortunately this schema is not perfect. First of all, point-to-point communication between all processes prohibits group communication or at best increases the group communication cost substantially, as a separate connection must be established with each of the members of the group individually. Totally synchronous communication between all objects is a restriction, though asynchronous protocols can be built on top of

synchronous primitives with extra costs. It eliminates the benefit of confining the local domain communication traffic, as the cost of connection establishment even between the neighboring nodes on the same domain, contributes to the overall system load unnecessarily. All these factors can contribute negatively to the predictability of the system. Therefore, the IPC has been designed and implemented with group communication in mind for the reasons that are explained in the following sections. The reasons why group communication protocols might be essential in a real-time system environment are elaborated as well, in the following sections.

The hardware testbed used in the initial phases of the implementation of the RT-DOS IPC services is shown in **Figure 5.4.2(1)**. The four Transputer boards that are connected together to form a *System Loop* and two application *Domain Loops* are *B004*, *B003*, *B006*, and *TMB12*.

B004 is a standard PC board plugged into an 8 bit PC slot, it hosts a T414 Transputer and it has a 2MB on board memory. This board is used for system development and all Transputer development system software also runs on this board. One of four links of T414 Transputer on the board is connected to the IBM PC system (mapped on PC RAM memory as a memory mapped device) and it interfaces (via C012 link interface hardware) with the PC keyboard and screen through its *Link0* link (Transputer links are numbered physically from 0 to 3). This board also serves as master Transputer board to reset all the daisy chained boards through its configuration link jumper.

Please note that, though the boards of the testbed which are involved in the research project did not change much through the implementation of the IPC, the numbers and types of Transputers that these boards hold changed according to the availability of Transputer modules (TRAMs) in time, as can be seen in **Chapter 7**.

After compilation and linking, network programs are downloaded through one of its links (in our case *Link0*) which is connected to the one of the next board's links.

All other boards are installed in a separate self-powered unit which has 12 slots for different Transputer boards. The B006 board (which has four T425 Transputers with 1MB of RAM installed on it) is connected to a VT220 terminal through a RS232 serial port which is available on this board.

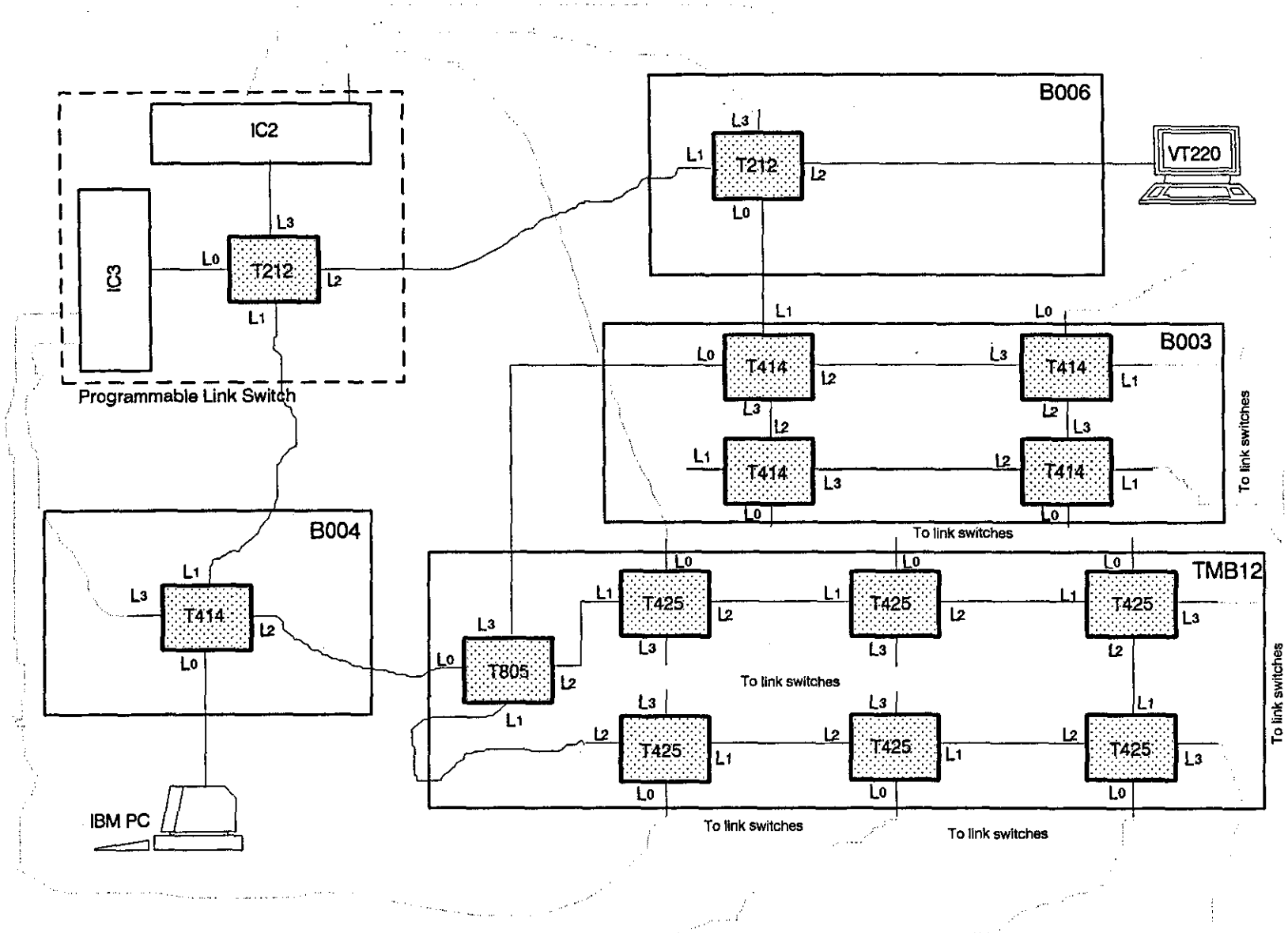


Figure 5.4.2(1) : RTDOS Testbed and Transputer Boards Used in IPC Implementation

This board has 1MB RAM for system development (it has been used with VAX/VMS version of OCCAM based Transputer Development System (TDS) for downloading the application code developed on VAX systems), and a 16 bit Transputer (T212).

The *B003* board has four T414 Transputers each with a 256KB of RAM connected through their *Link2* and *Link3* links to each other. *Link1* and *Link0* of each Transputer has been taken out to special edges to be able to connect them to any link of other Transputers through standard Inmos link jumpers. The boards *B003* and *B006* are connected each other through these edges using *Link1* of a Transputer on *B003* and *Link0* of T212 using these jumpers. All other unused links of Transputers are connected to a programmable link switch hardware which is the physical backbone of the RT-DOS IPC mechanism. As the switchboard deserves special consideration, it will be explained in more detail in the following section.

TMB12 is a board with 10 Transputer TRAMs installed on it each one with a 256KB of RAM. One of the Transputers, so called Root Transputer, is T805 and all others are T425 (all 32 bit). The Root Transputer is connected to the *B008* board through its link *Link0*, to the *B003* board thorough its link *Link3*. All unused links of Transputers on the *TMB12* are connected to the programmable link switches (two *C004* controlled by a T212 Transputer) which reside on the same board, so that they can be connected to any other Transputer (which is connected to the switchboard) on the network on fly dynamically. Though the link switch hardware physically resides on the *TMB12* board, it is shown as an abstract unit in the figure. A T212 Transputer also is installed on the *TMB12* board as a configurer Transputer to control two *C004* link switches, to dynamically connect the links of programmable switch for establishing physical circuits between the processes locating on different nodes.

In **Figure 5.4.2(1)**, dotted lines indicate the indirect physical connections between Transputers established via *C004* link switches, while darker lines indicate direct (from one Transputer to another neighbor Transputer) link connections between Transputer via either hardwired circuitry on boards, or using Inmos jumpers length ranges between one inch to a few meters.

A possible configuration (locations) of the IPC related RT-DOS system servers (assuming that **Figure 5.4.2(1)** is used as hardware topology) are shown in **Figure**

5.4.2(2). The T212 Transputer located on *B006* board is used as a node which hosts the *Terminal Server* process, while the *B004* Transputer is used as the *Name Server* and the *File Server* node. System initialization and downloading of the configuration tables which contain information about the connectivity features of each Transputer on the network are also carried out through this node as only it has the interface with the outside world for program/file loading interactively or in batch mode. In the figure, the *System Loop* is formed by the Transputers connected through a dark line, while the two domain loops are formed by connecting Transputers via a dark dotted line. The *Application Domain Loops* and the *System Loop* intersect on the *Domain Manager* Transputers.

In our testbed, the *System Loop* consists of the *Connection Server* node, the *Name Server* node, the *Terminal Server* node, and the two *Domain Manager* nodes. One application *Domain Loop* is formed on the *B003* board and consists of four Transputers including a *Domain Manager* node. Another application *Domain Loop* is formed on the *TMB12* board and consists of seven Transputers including the *Domain Manager* node. A number of user processes are randomly distributed to the different nodes of domain loops, continuously generating messages to be sent to each other either through the point-to-point connections (requires physical link connection establishment between related Transputer links) or datagram services. Using different combination of process topologies, a varying range of message traffic is generated to measure the performance changes of the IPC mechanism under different communication load conditions. In the figure, the arrows indicate the direction of message traffic in the loops.

5.4.3 C004 Dynamic Link Switches

The Inmos communication link is a high speed system interconnect which provides full duplex communication between members of the INMOS Transputer family, according to the INMOS serial link protocol. The *IMS C004*, a member of this family, is a transparent programmable link switch designed to provide a full crossbar switch between 32 link inputs and 32 link outputs [InmosJ 1991]. The *IMS C004* can switch links running at either 10 Mbits/sec or 20 Mbits/sec. It introduces, on average, only a 1.75 bit time delay on the signal. Link switches can be cascaded to any depth without loss of signal integrity and can be used to construct reconfigurable networks of arbitrary size. The switch is programmed via a separate serial link called the *Configuration Link*.

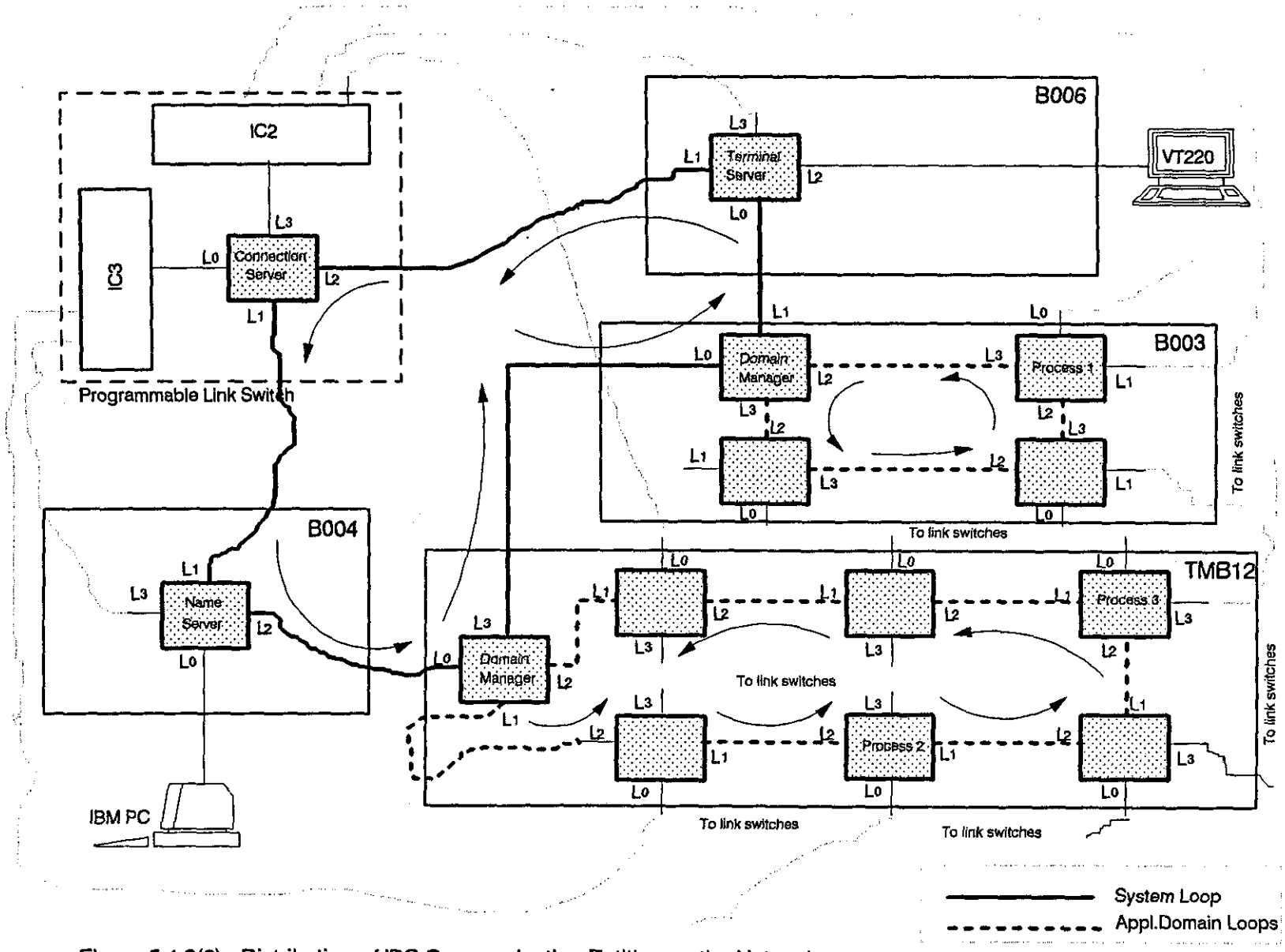


Figure 5.4.2(2) : *Distribution of IPC Communicating Entities on the Network*

Products of different performance (link speed) can be interconnected directly with these switches. *C004* link switches can also be interconnected to each other to form virtually unlimited switches. **Figure 5.4.3(1)** shows a network of 32 link switches controlled by one Transputer through a master switch which is, in turn, connected to all the slave switches. Here, all the Transputers which are connected to any of these switches can establish physical point-to-point communication links with any other Transputer on the network, through these switches. In the figure dotted lines indicate link connections set through the switches between Transputers.

Another switch configuration, in which each *C004* link switch is controlled by a separate Transputer that is chained to a pipeline of control Transputers, is shown in **Figure 5.4.3(2).** In this configuration, management of each physical subnetwork of links is separated to increase concurrency of switch hardware and to eliminate single control point bottleneck. But coordination of distributed link configuration software agents is a prime issue to be handled.

In the RT-DOS testbed, two *C004* link switches are physically located on *TMB12* board and both are controlled by a T212 Transputer concurrently, which is connected to both of them via its links 3 and 0. All Transputers on the network, regardless of their physical board location, are connected to these two link switches (through their free links), via special INMOS jumpers.

The IMS *C004* links implement the following subset of the configuration messages:

[*0*] [*input*][*output*]	Connects *input* to *output;*
[*1*] [*link1*][*link2*]	Connects *link1* to *link2* by connecting the input of *link1* to the output of *link2;*
[*2*] [*output*]	Inquires which input the *output* is connected to. The *IMS C004* responds with the input. The most significant bit of this byte indicates whether the output is connected (bit set high) or disconnected (bit set low).

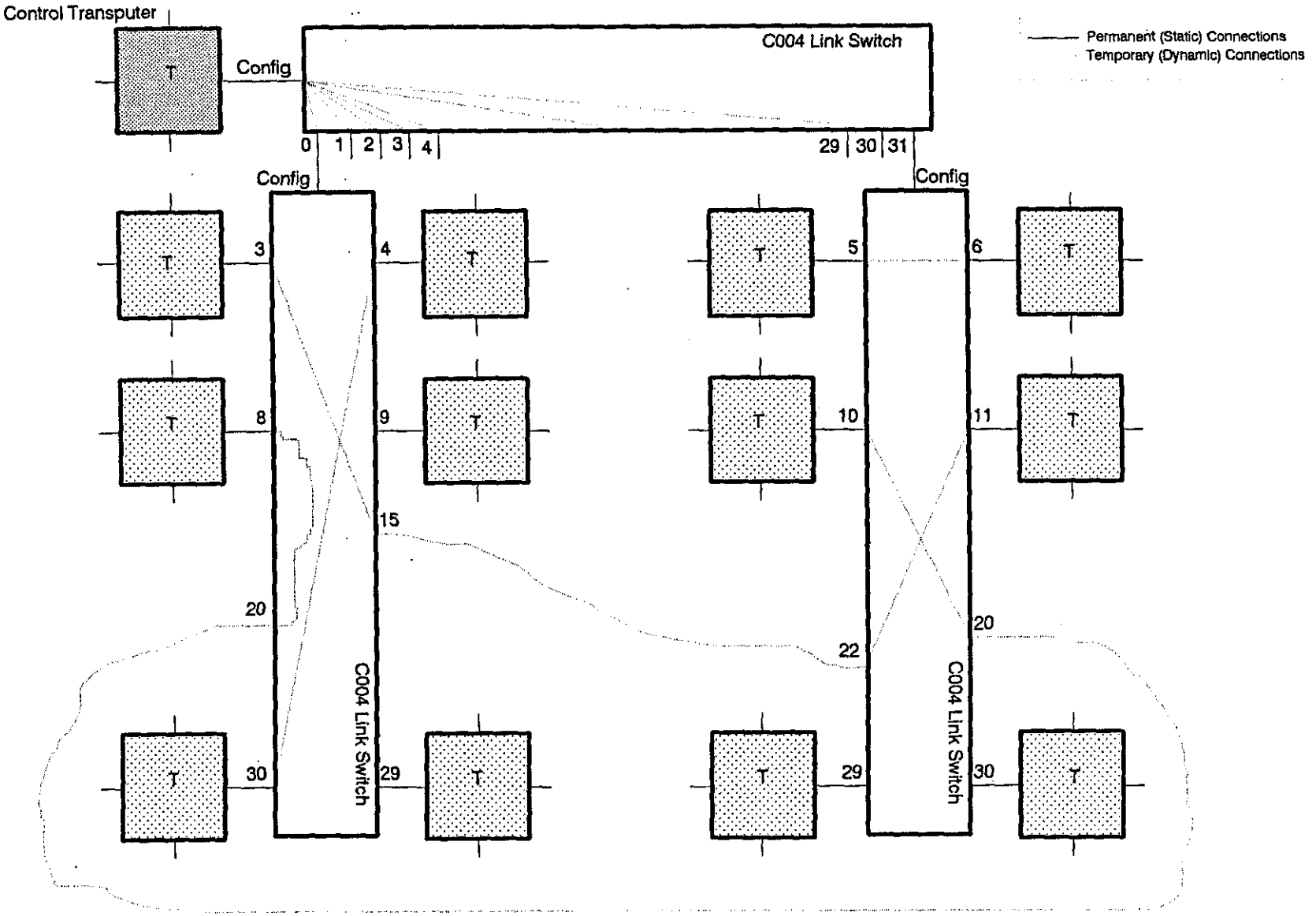


Figure 5.4.3(1) : Network of C004 Link Switches Controlled by One Transputer

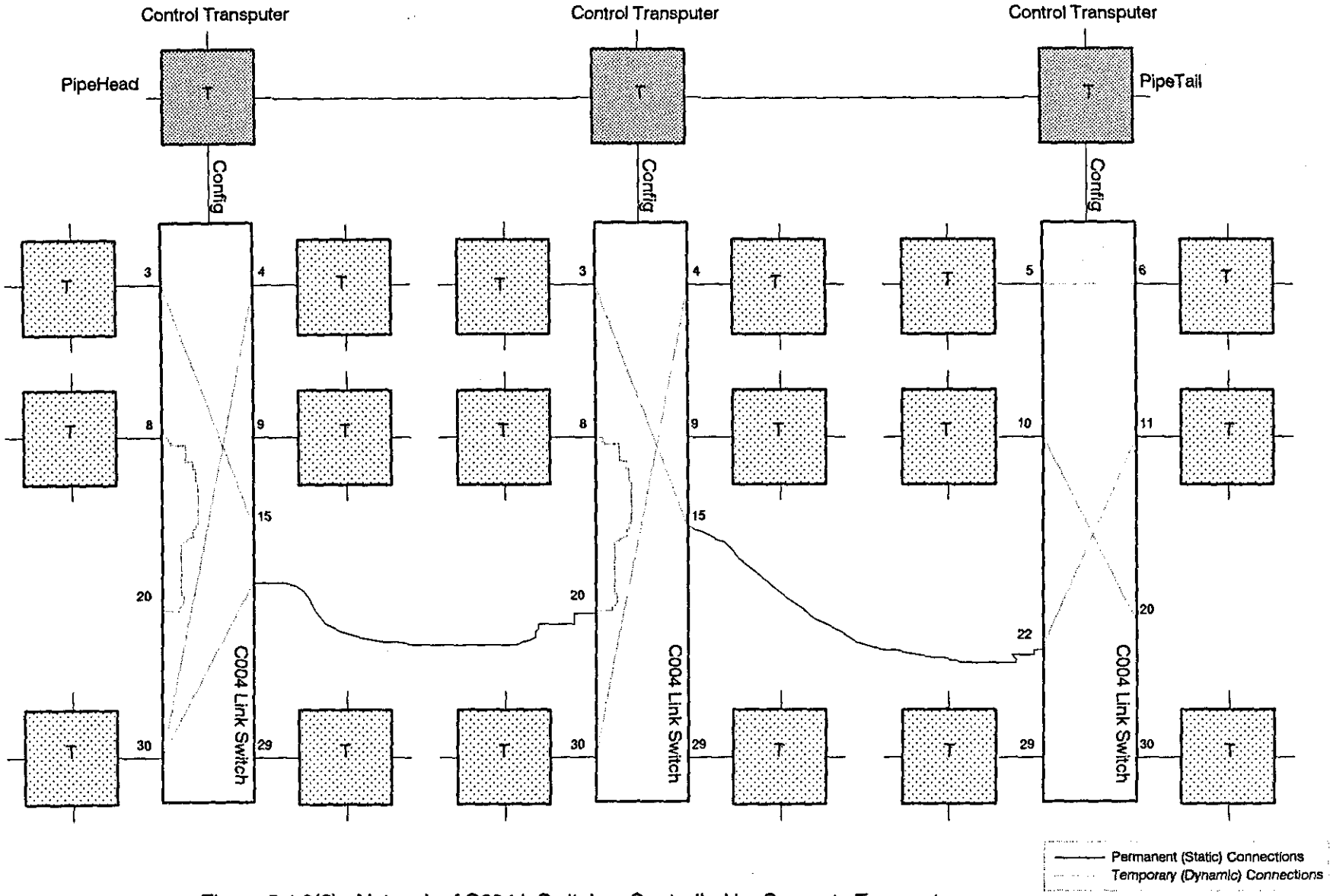


Figure 5.4.3(2) : Network of C004 L.Switches Controlled by Seperate Transputers

[3]	This command byte must be sent at the end of every configuration sequence which sets up a connection. The *IMS C004* is then ready to accept data on the connected inputs;
[4]	*Reset*s the switch. All outputs are disconnected and held low. This also happens when **Reset** is applied to the *IMS C004*;
[5] [*output*]	Output *output* is disconnected and held low;
[6] [*link1*][*link2*]	Disconnects the input of *link1* and the output of *link2*;

When *Reset* is applied to the *IMS C004* the outputs are disconnected. After power is applied and before any configuration message is transmitted to the *IMS C004*, a software reset byte (control byte [4]) must be sent. This has the effect of disconnecting the outputs.

The *IMS C004* is internally organized as a set of thirty two 32-to-1 multiplexors. Each multiplexor has associated with it a six bit latch, five bits of which select one input as the source of data for the corresponding output. The sixth bit is used to connect and disconnect the output. These latches can be read and written by messages sent on the configuration link via *ConfigLinkIn* and *ConfigLinkOut*.

The output of each multiplexor is synchronized with an internal high speed clock and regenerated at the output pad. This synchronization introduces, on average, a 1.75 bit time delay on the signal. Since the signal is not electrically degraded in passing through the link switches (as it was stated before), it is possible to form point-to-point links between remote nodes through an arbitrary number of link switches with a little communication performance overhead.

Each input and output is identified by a number in the range 0 to 31. A configuration message consisting of one, two, or three bytes is transmitted on the configuration link.

The configuration messages sent to the switch through this link have been given above. For a full hardware description of *C004* switch please refer to [InmosJ 1991].

Transputers physically located on different hardware boards and other platforms can be connected to each other through these *C004* link switches to form logical loops as shown in **Figure 5.4.3(3),** and can be dynamically reconfigured to adapt to changing requirements of application systems which are running on these platforms. The RT-DOS kernel is one of these applications which has been built on these facts to get benefit of the flexibility provided by *C004* switches. The RT-DOS IPC mechanism is also implemented on these loop topologies to exploit the given benefits.

5.4.4 The IPC Layers

The RT-DOS kernel interprocess communication (IPC) subsystem is built around a three-layer network architecture as shown in **Figure 5.4.4(1).** These are:

- The *Physical Layer*;
- The *UDL* (unreliable datagram) *Layer*; and
- The *IPC Layer*.

A comparison between the *ISO OSI* network reference model [TanenbaumE 1981] protocol layers and the RT-DOS IPC layers can also be seen in the figure. The IPC implements the functionality of the first four *OSI* layers. It is to be noted that there is no attempt to adopt the *ISO OSI* layers either in terms of protocols or message frame formats. The comparison is only in terms of services that are provided to the users of the IPC mechanism.

The *Physical Layer* is related to the transmission of stream of bits over a point-to-point communication channel. The issues here are largely dealing with mechanical, electrical and procedural interfacing of nodes to the subnet and to each other. Transputers implement physical level protocols at the hardware level. In fact Transputers have a micro coded firmware level kernel which can manage short term process scheduling, as well as handling the signal equalization problems of four serial links. Some part of the *Data Link Layer* of the *OSI* model, which is related to transmission of data frames from one node to another without errors (data loss or duplication), is also implemented by this micro kernel that starts running on each Transputer after a reset operation.

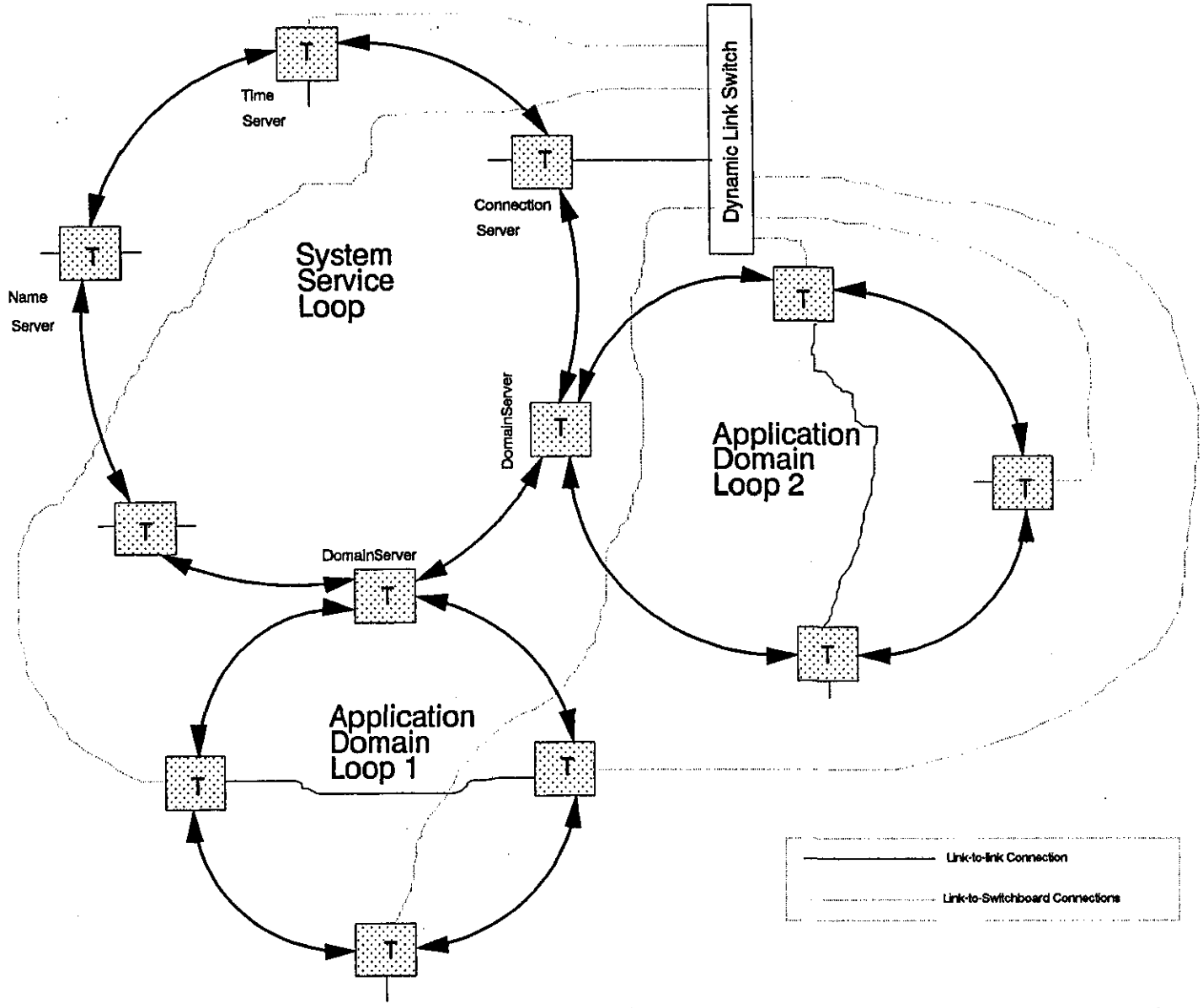


Figure 5.4.3(3) : Transputers on Different Loops Connected By C004 L.Switch

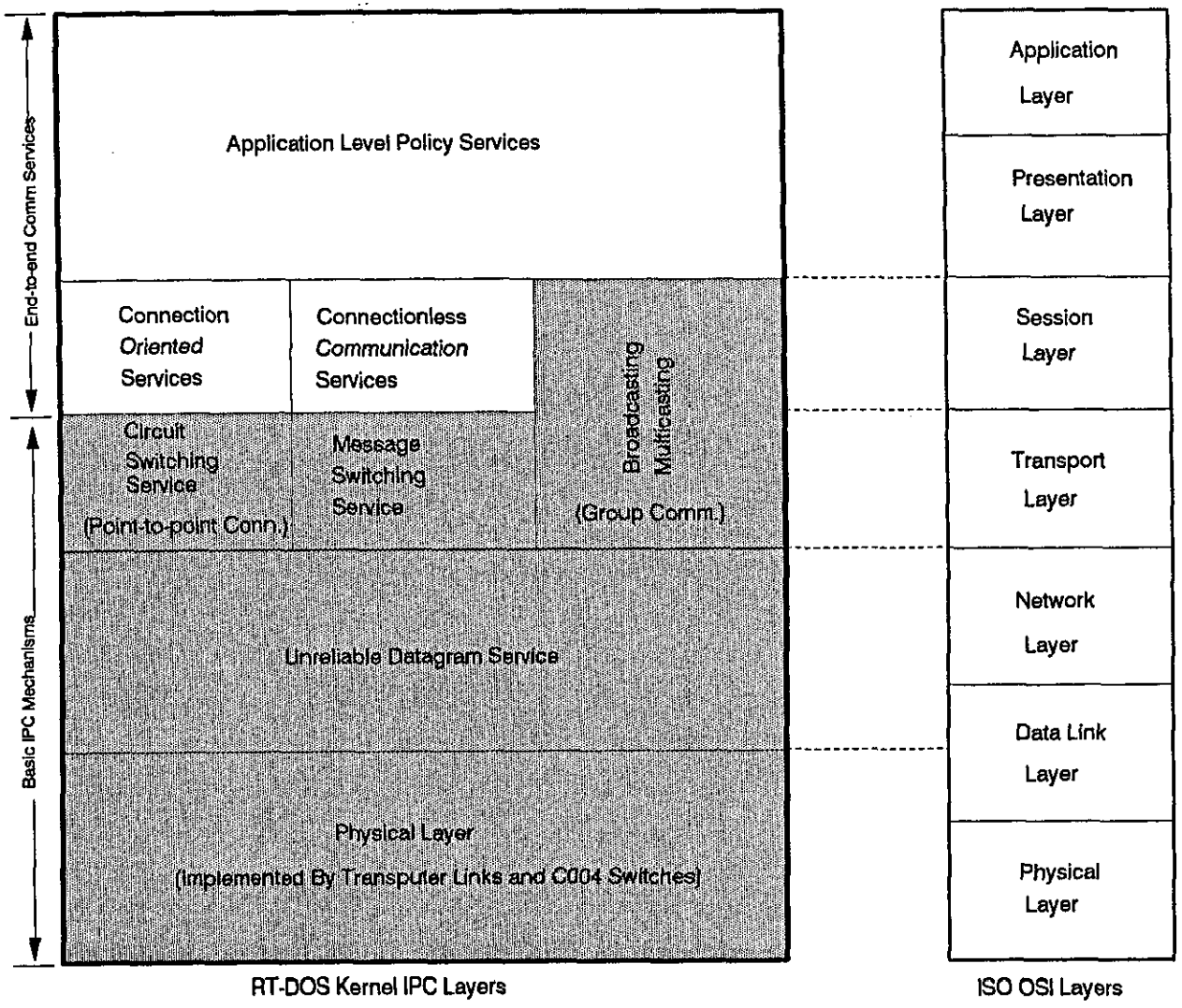


Figure 5.4.4(1) : RT-DOS Kernel IPC Layers versus ISO OSI Layers

Transputers (which have been designed to implement the CSP based OCCAM communication model) automatically support CSP model communication which is based on rendezvous semantics [Hoare 1978], at the hardware level. Data transfer between two Transputer links is synchronous, and both parties must be ready and waiting to communicate, for data transfer to occur successfully. An important assumption of current Transputer implementation of the CSP model (OCCAM for instance) is that communicating processes are expected to be located either on the same Transputer (communication channels are mapped onto a memory location) or on neighboring Transputers (communication channels are mapped onto Transputer links). For a complete physical network location transparency of communicating processes, which is one of the design considerations of the RT-DOS kernel and its IPC mechanism (to support dynamic process migration for fault tolerance, and system reconfiguration), neighborhood of communicating processes must not be assumed a priori.

The RT-DOS IPC extends the Transputer implementation of the CSP model [Hoare 1978] in a number of ways. Firstly, it pushes the native transparency definition from physical neighborhood to total network transparency. Any process on the network can establish a point-to-point synchronous data transfer channel with any other process on the network. The *Circuit Switching Service* layer depicted in **Figure 5.4.4(1)** implements this protocol as is explained later in more detail. Secondly, the RT-DOS IPC mechanism introduces the N:1 communication model to encompass the client-server paradigm. However, without group communication support at the *IPC Network* level, the *N:1* model would be unacceptably inefficient, therefore the IPC has been designed with the group communication (broadcasting, multicasting, domain casting, etc.) support in mind from its initial stages. The *Broadcasting Multicasting* layer in the figure implements these group communication protocols. The *Message Switching Service*, shown in the figure, is related to the connectionless *1:1* communication protocols (i.e., protocols in which senders do not request any replies from the recipient processes for their messages).

All IPC services except the *Circuit Switching Service* are implemented as unreliable at the IPC level, because of the idea that the IPC mechanism should be compact and minimum as it is replicated at each node on the network.

Though it provides a rich variety of the communication services, it assumes that the semantics of the communication is determined by the upper layer protocols built on top of the IPC mechanisms. The IPC provides a communication path (transport service) between parties, using the connection protocol (circuit switching, message switching, or group communication protocols) which is chosen by the policy layers of the kernel considering user service requirements. The fact that Transputer links are highly reliable at the physical level, also supports the idea of handling error checking and recovery in terms of overall system communication performance. This provides the user of the IPC services with the flexibility of choosing an appropriate cost/performance level. If a process needs to notify a group of processes and it doesn't expect any feedback from the recipients, it doesn't have to set a very costly point-to-point connection with each recipient individually. Broadcasting a message will be the most convenient and cheapest option. It is even worse depending on only the point-to-point communication, if the sender doesn't know (obviously, doesn't need to know either) all the recipients and their addresses on the network.

The *Unreliable Datagram Service* is the backbone of the IPC and its upper level protocols. Regardless of the type of communication protocols they are using, all the transport level services use the datagram service. It provides the RT-DOS Kernels, with a simple transport mechanism from the prespecified source node to the destination node, regardless of the meaning or contents of messages that it carries. The *Network Layer* functions of the *OSI* model are included in the unreliable datagram service, i.e. network routing and addressing. The circuit switching and message switching services, including group communication options such as broadcasting and multicasting, are built on top of this datagram service. Datagram packets are of fixed length for control messages used between kernels to establish connections, but of variable length in size if they carry real messages (data) between distant processes. Packet formats and their features, such as length, fields, valid field values together with the meaning of each one will be discussed later in **Sections 5.4.6** and **5.4.7**.

Each of these service layers interface with other layers by using a number of primitives which are provided by the lower layers, to implement the related protocol between service layers. **Figure 5.4.4(2)** summarizes the basic primitives provided by each service layer to the upper ones, and the primitives which are provided to them by the lower layers in turn.

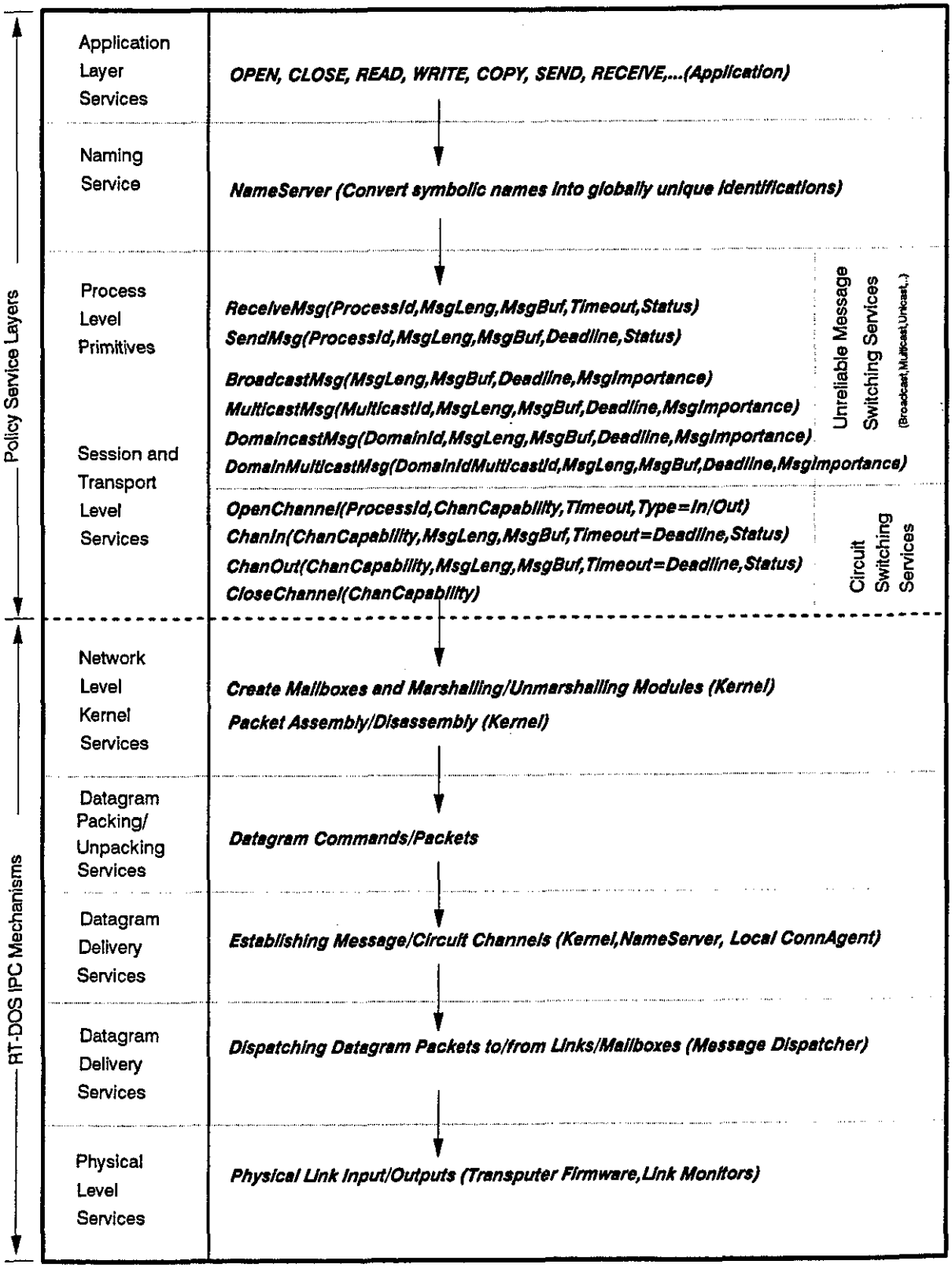


Figure 5.4.4(2) : Interfaces of IPC Layers With Kernel and Related Parties

At the uppermost level, applications request their services by issuing commands such as *OPEN*, *CLOSE*, *READ*, *WRITE*, *COPY*, *SEND*, and *RECEIVE*. At the second level, compilers will generate related code to perform calls to the name service routines for converting application level symbolic names into system wide global unique object identifiers, especially process, node and domain identifications. Application level interfaces and related issues are not covered at the IPC level, and will not be discussed in this thesis. At the IPC level, all symbolic object names are assumed to be converted into unique identifications by a distributed naming service, as will be explained in **Section 5.4.5.2**. Using parameters, applications specify their requirements: the importance of processes and messages (priorities); time fences for the validity period of messages; type of communication (whether a reply expected or not); reliability expectations; size of data to be exchanged; list of message senders or receivers (for group communication and client-server type protocols); and of preferred protocol (synchronous-circuit switching or asynchronous-message switching).

Process level primitives are receive message (*ReceiveMsg*), send message (*SendMsg*), broadcast message (*BroadcastMsg*), multicast message (*MulticastMsg*), domain cast message (*DomaincastMsg*), multicast message in a domain (*DomainMulticastMsg*), open a channel (*OpenChannel*), read from channel (*ChanIn*), write to channel (*ChanOut*), and close channel (*CloseChannel*).

The last four primitives (*OpenChannel*, *ChanIn*, *ChanOut*, and *CloseChannel*) are related to establishing reliable circuit switched channels (point-to-point connections) between process pairs. *OpenChannel* opens a logical channel between two processes by establishing a physical connection through physical links of two non-neighboring Transputers on which communicating processes are residing. Each one of the corresponding processes issue this request to inform the local kernel about their intention to communicate with the other parties synchronously. *ChanOut* and *ChanIn* are issued by the sender and receiver processes respectively, to write and read data to/from the established channel. *CloseChannel* is issued by the client (initiator of the communication). The *Circuit Switching Service* will be discussed in **Section 5.4.6.2** in more detail.

The rest of the process level primitives are related to unreliable communication services and are implemented as part of the datagram service. *ReceiveMsg* and *SendMsg* are used for unicast (one-to-one) communication between two processes. One-to-one communication is referred to as a packet switching service and is discussed in **Section 5.4.6.3**. The rest of the primitives are related to group communication. The *BroadcastMsg* is used to broadcast a message to all nodes on the network. It is usually issued by a kernel to locate a process address through the network. The *MulticastMsg* is used to distribute a message copy to a group of processes related to the same task, such as time servers (for updating global time), displaying a status message or statistics on a group of operator terminals, or updating a group of replicated copies of files. The *DomaincastMsg* is similar to the *BroadcastMsg* in terms of functionality except that its context is a specific domain rather than whole network. The *DomainMulticastMsg* is also the same as the *MulticastMsg* in terms of functionality, but it addresses a group of processes in a specific domain. The last two group communication primitives are implemented for efficiency reasons only, to reduce the cost of message broadcasting by confining local message traffic within domain boundaries. Group communication primitives, their objectives and benefits are discussed in **Section 5.4.6.4**.

Process level primitives are implemented at datagram level as a group of datagram commands. Network level routing, packet assembly/disassembly, handling of related data structures (look up tables, message, buffers, ports and mailboxes, etc.), and realization of semantics of each protocol are defined at the datagram level with the collaboration of the kernel. These services are summarized as network routing, datagram packing/unpacking, and datagram delivery services. The *Circuit Establishment Protocol* is also implemented at this level.

The last layer of the RT-DOS IPC mechanism is the physical level link connection service. Though once two Transputer links are physically connected synchronization and transfer of data through link pairs are handled by Transputer hardware, a policy layer should map channel names established to connect two distant processes, to a pair of physical links and connect them to each other through *C004* programmable link switches. Data exchange between process pairs is implemented via Transputer link I/O commands (*LinkIn*, *LinkOut*) at this lowest level. Packet forwarding from one node to

the next one through the network is also implemented by this physical link I/O commands.

5.4.5 The Basic IPC Entities and the Communication Primitives

The basic RT-DOS IPC objects (communicating entities) are processes, the IPC servers, the RT-DOS Kernel, and the *Name Server*. All system servers, including the *Name Server*, are treated as ordinary processes by the RT-DOS IPC mechanism. The IPC provides them with a message transport service, regardless of their functionality, importance, or the level of layer in which they reside. The semantics of communication between any communicating parties, error checking, recovery and communication control are assumed to be implemented at process level (end-to-end).

Primitives used to implement different communication protocols are summarized in **Figure 5.4.4(2)**, and they are discussed in the relevant paragraphs of **Section 5.4.6**. The IPC *Connection Service* consists of two agents and communication stubs created by each kernel on behalf of the communicating processes. The IPC *Connection Manager* and *Switch Manager* agents are explained in **Sections 5.4.5.3** and **5.4.5.4**, respectively.

5.4.5.1 User Processes and System Servers

All operating system servers, including the ones which form the kernel policy layers, are treated as ordinary processes, and all of them use the IPC services to communicate with each other by message exchange. For example, the *Time Server* implements a network wide global timing service, and uses the IPC message passing mechanism to update (synchronize) clocks on all nodes. The *File Servers* and the *Terminal Servers* are also treated in the same way. The IPC message passing mechanism transfers messages between ports of processes. For multiport servers (for example, the *File Server*), *N:1* semantics of communication and client-server protocols are implemented at process level.

One of the most important system servers is the *Name Server* and, as naming of communication entities are directly related to interprocess communication, it is discussed below.

5.4.5.2 Name Server

Name servers in general convert symbolic object names into often unique numeric identifications. The RT-DOS *Name Server* also converts the RT-DOS object names (ports, channels, files, processes, servers) into system wide unique identifications (32 bit capabilities) which are generated from a partitioned name space. The RT-DOS has only one predefined well known name which is the *Name Server* itself. All other names are converted into unique identifications dynamically by the naming service. To convert object identifications into addresses, the sender kernel broadcasts a "*locate address*" command and the kernel of the destination object replies with the location information. Each kernel maintains a local address caching mechanism to keep physical locations of the last referred objects, to increase the IPC performance. The penalty incurred by dynamic name binding is offset by the advantages of location independent addressing and the ability to reconfigure the system. That is, system objects can migrate dynamically during execution. None of the system servers except the *Name Server* has a predefined address or identification. This provides the flexibility of changing locations of system services (as well as application processes) according to the requirements of a specific application.

Processes, system nodes, and application domains have system wide unique identifications, created using a centralized name service. The rest of the IPC objects are created with locally unique identifications. For example, mailboxes are created locally by kernels and global uniqueness for point-to-point connections is established by combining them with port, node and process identifications.

Figure 5.4.5.2(1) depicts a simple name binding table structure used by the naming service. Globally unique identification of each process, node, and domain are kept in this centralized table. Each kernel also keeps track of the physical locations of those objects to which local kernel accesses can be made. These address resolution tables are maintained dynamically. If the location of any object (process, etc.) is not found in the local address resolution table when it is referred to by a local process or kernel, the local kernel broadcasts a "*locate address*" datagram through the network. Then, the destination kernel sends a unicast datagram to the originator of the broadcast message, with the address of the destination object. Then the originator's message is sent to the destination address directly, and the local address resolution table is updated for later

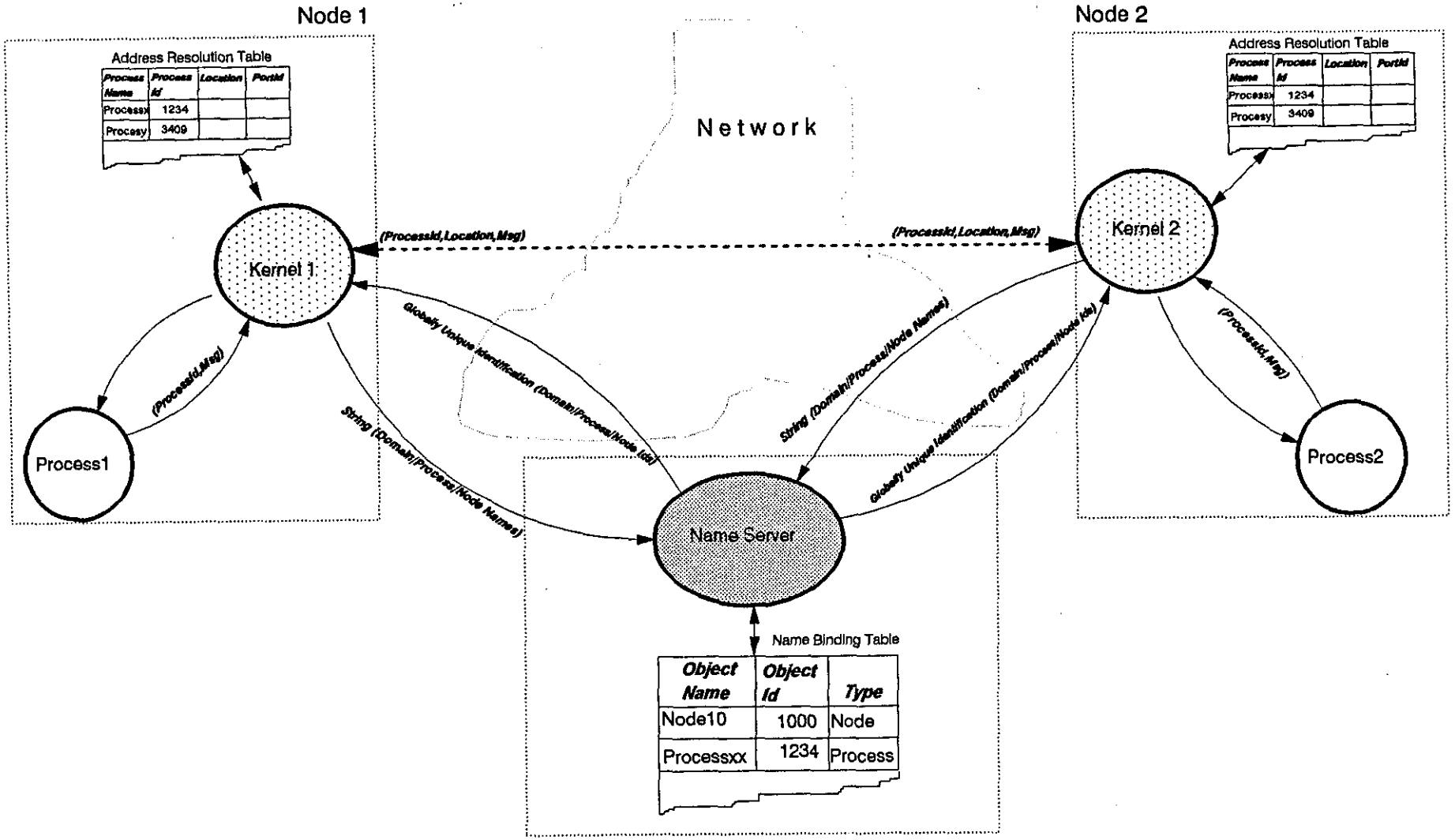


Figure 5.4.5.2(1) : RT-DOS Name Server Dynamic Name Binding Protocol

reference. If the destination object migrates after the local cache table is updated, the former kernel of the migrated object sends an "*address changed*" notification to the originator kernel, so that it can broadcast a new locate address datagram to find the new address of the destination object. A message is sent to the destination with process identification and location address (combination of domain identification and node identification in domain). The mapping of process identifications into port capabilities is carried out by the destination kernel.

Object identifications are created using a random number generator to enable the reuse of the object ids for limiting the size of object name space (and consequently, reducing the size of the IPC datagrams). If a new identification conflicts with an existing one, another number is generated until its uniqueness is guaranteed. Locations are only known and used by kernels, and are totally invisible to the rest of the system. Assignment of domain and Transputer numbers is performed during system initialization, or following dynamic reconfigurations.

There are some positive and negative affects of generating object identifications serially and randomly on process migration. Nevertheless, the IPC mechanism is designed as a compact transport mechanism carrying messages between kernel entities irrespectable of how their identifications are created, and hence, a further elaboration of the maintenance of object identifications would be too much detail in this context. As the IPC mechanism assumes the uniqueness of identifications of the communicating objects, and maintains only address resolution tables to locate these objects when it is required, a full discussion of the details of the name server implementation is out of the scope of this thesis.

5.4.5.3 Connection Service Managers

The *Connection Service* (point-to-point communication service), one of the transport layer communication services, is built on top of the unreliable datagram service, and establishes end-to-end reliable connections over the switched Transputer links, using a rendezvous based strategy.

The connection service consists of three service agents: the *Connection Manager*, the *Switch Manager*, and the *Switcher*. Though connection service hardware and related components are physically distributed all over the network, logically it is a centralized service for each *System Loop* (more than one system loops per network can be established by gateways) and related application domains. The number and location of these agents are determined by the requirements of the supported real time systems.

The *Connection Manager* accepts connection requests issued by sender and receiver processes to establish a point-to-point connection between their address spaces through physical links of related nodes. It maintains a request matching table and defines the node pairs (Transputer pairs) to be connected matching related requests. It then conveys its decision to the *Switch Manager*, so that it can choose the available links of nodes to be connected. Mapping of the physical links to switchboard edges is carried out by the *Switch Manager* using a configuration table which is created during the system initialization process as part of the network loading. It also notifies the requesting processes about the connection status, and sends disconnection requests (connection release requests) to the *Switch Manager*.

The *Switch Manager* accepts connection requests from the *Connection Manager* and determines the links to be connected. It maintains a configuration table which is created during system initialization, and keeps track of available links at each node for connection establishment. After each successful connection establishment, it informs the *Connection Manager* which, in turn, notifies requesting processes. The *Switch Manager*, after deciding about the links to be connected at each node, sends a requests to the *Switcher* to execute the hardware commands to set the physical connection between the node links, through programmable link switches. Before sending link configuration commands to the *Switcher*, the *Switch Manager* maps Transputer link numbers into switchboard link numbers using the link configuration table.

The *Switcher* is a simple process waiting for link configuration commands issued by the *Switch Manager*. It accepts one of the seven *C004* link switch configuration commands, and sends them to the link switch hardware through its config link which connects the link switch hardware to the Transputer on which the *Switcher* resides.

Data structures used by these servers to set connections and all steps of a connection establishment protocol are explained in **Section 5.4.6**.

5.4.5.4 Communication Agents

The *Communication Agents* monitor communication operations of ports and mailboxes in each kernel. They keep track of the status of connections and message buffers for recovery and restart procedures, in coordination with the datagram delivery service.

5.4.6 Message Exchange Protocols Used Between Communicating Entities

The RT-DOS processes communicate with each other using two basic message passing protocols. One of them is direct data transfer between process address spaces using a reliable synchronous point-to-point physical connection. This method is discussed as the *Circuit Switching Service* in **Section 5.4.6.2**. This method is used only for bulk data transfer between processes, and for *N:1* type of client-server communications to reduce the connection establishment costs which are explained later on.

Another communication method is a group of asynchronous unreliable message exchange protocols used for the transfer of small messages, especially between processes communicating in the same domain to retain local message traffic in the domain boundaries. These communication protocols are considered as the *Packet Switching Service* and explained in **Section 5.4.6.3**.

The group communication services are integrated into the packet switching service and handled at this level. These services (broadcasting, multicasting, domaincasting, and domain multicasting) and circuit switching service are all implemented on top of an unreliable datagram service. Therefore, the unreliable datagram service is discussed first in **Section 5.4.6.1**. Though group communication protocols are implemented within the message switching service, they are separately discussed in **Section 5.4.6.4**.

5.4.6.1 Unreliable Datagram Service

The core of all the IPC services is a simple unreliable datagram service. It carries control commands between kernels for connection establishment, as well as short messages between processes. A simple message routing algorithm forwards datagram packets from node to node in the *Domain Loops* or the *System Loop*. Each packet

carries full address information of the packet source and destination (if the packet is not a "*locate address*" command).

The packet routing algorithm behaves slightly differently at different nodes, though the same copy of the code is running on each one. A domain manager node is an intersection between an application *Domain Loop* and the *System Loop*, and thus it might direct a packet through the *System Loop* (if destination domain is different than the current domain) or to the current application domain loop (if destination and current domains are the same). As all four links of the *Domain Manager* nodes are dedicated for the loop connections (two for application *Domain Loop* and two for the *System Loop*), there is no link available for a connection establishment with another node.

Figure 5.4.6.1(1) depicts the architecture of the datagram routing service on a *Domain Manager* node. A datagram packet can arrive at the node from one of two links (*FromOsNode* and *FromDomainNode*). *FromOsNode* link connects the node to the *System Loop* and packets coming through the *System Loop* enter the node from this link. A line driver process waiting for arriving packets from the link puts the incoming packets on a queue (*FromOsQ*). Another process (*FromOsDispatch*) takes the packets from the queue and checks with the destination node. If the destination node is not the current node, then the process checks with the destination domain. If the destination domain of the packet is different than the current one, then the packet is put on *ToOsQ* queue from which packets are sent to the next node of the *System Loop* by *ToOsLink* process. If the destination node is the current node, then the packet is put on the queue *FromOsToNodeQ*, to transfer to a local process by the *NodeDispatch* process.

The second data link from which datagram packets arrive is *FromDomainNode* link. The line driver process *FromDomainLink* accepts packets coming through this link from the application domain loop and puts them on the queue *FromDomainQ*. Another process (*FromDomainDispatch*) takes the packets from the queue and puts on the queue *FromDomainToNode* if the destination domain is different to the current one, or the destination node is the current node.

Packets put in this queue are distributed to either local processes (if destination node is the current one), or sent to the *System Loop* through *ToOsQ* queue (if the destination domain is different to the current domain).

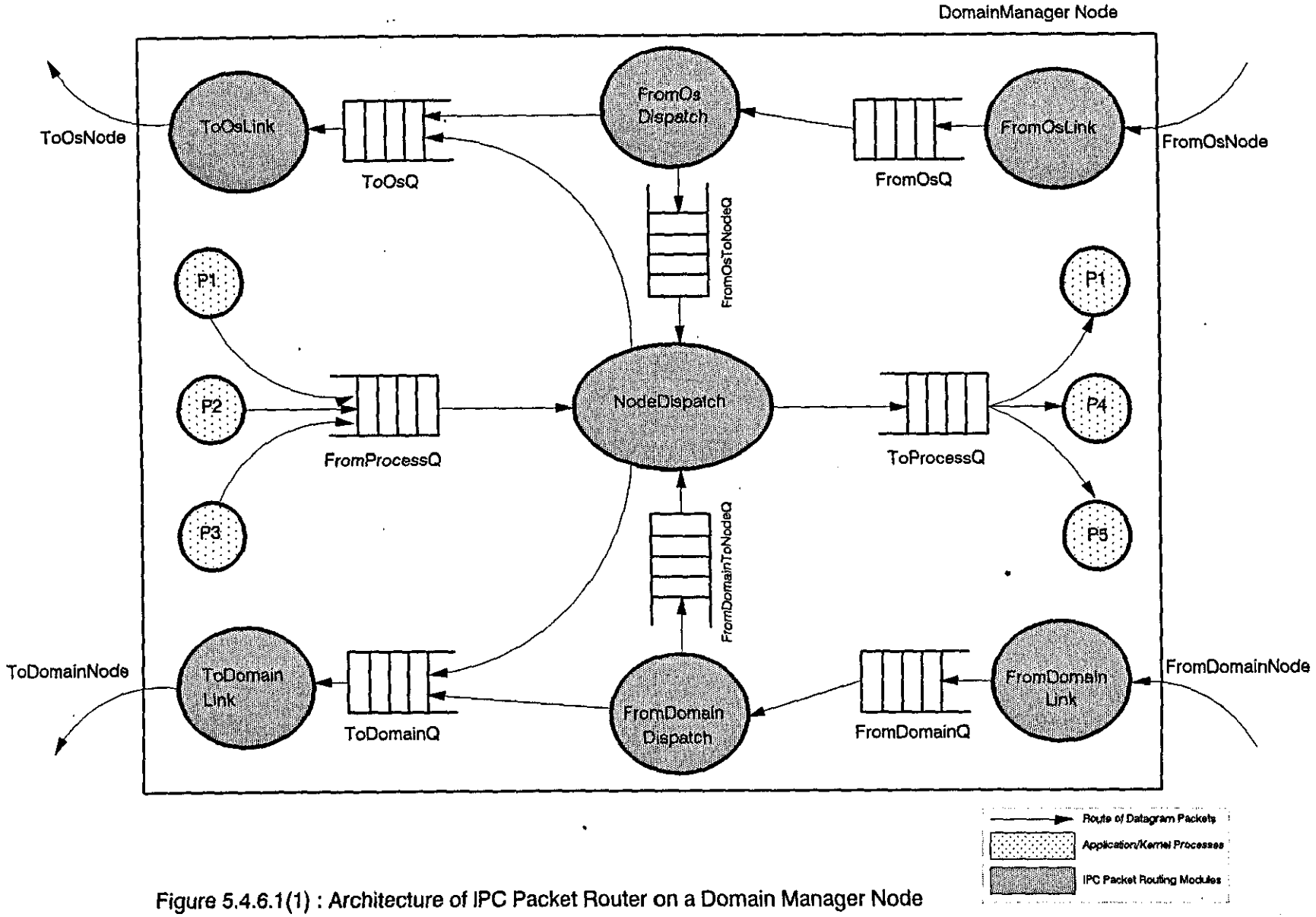


Figure 5.4.6.1(1) : Architecture of IPC Packet Router on a Domain Manager Node

If the destination domain is the current one and the destination node is not the current node, then *FromDomainDispatch* process puts the packet on *ToDomainQ* queue from which packets are sent to the next domain node through the link *ToDomainNode* by the process *ToDomainLink*.

The *NodeDispatch* process reads packets from the queues *FromProcessQ* (packets sent by local processes), *FromDomainToNodeQ* (packets sent by other nodes of the current domain to nodes on other domains), and *FromOsToNodeQ* (packets coming from other domains and targeted to the current domain or node); and then distributes them to the queues *ToProcessQ* (destination process of packet is one of the local processes), *ToOsQ* (destination domain of the packet is not the current one), or *ToDomainQ* (destination domain of the packet is the current one but the destination node is not the current node).

Packets of local processes and remote processes are treated in the same way. From the implementation point of view, packets are copied twice only from input link to the kernel buffers and from the kernel buffers to the output links or local process buffers. All queues are lists of packet buffer addresses only.

The datagram packet routing algorithm is slightly different at *Os/Domain* (non-domain manager) nodes, though the same copy of the code is executed on all nodes. As can be seen in **Figure 5.4.6.1(2),** some modules of the datagram delivery service are suppressed on *Os/Domain* nodes. *Os/Domain* nodes are connected only to one loop (either the *System Loop* or the *Domain Loop*), and datagram packets can come through a domain link only (*FromDomainNode*). Outgoing packets also can be sent through a domain link (*ToDomainNode*) only. *FromOsLink*, *FromOsDispatch*, and *ToOsLink* processes are not active; so that the *NodeDispatch* process accepts packets from the local processes and the *FromDomainDispatch* process, and then distributes them to the local processes only.

The main difference between domain manager and *Os/Domain* nodes is in their functionality. *Os/Domain* nodes have two free links to be used in point-to-point data transfer between processes residing on different nodes. The datagram service carries control commands to establish a connection between these nodes only, but exchange of data after setting physical connection takes place independently of the datagram services.

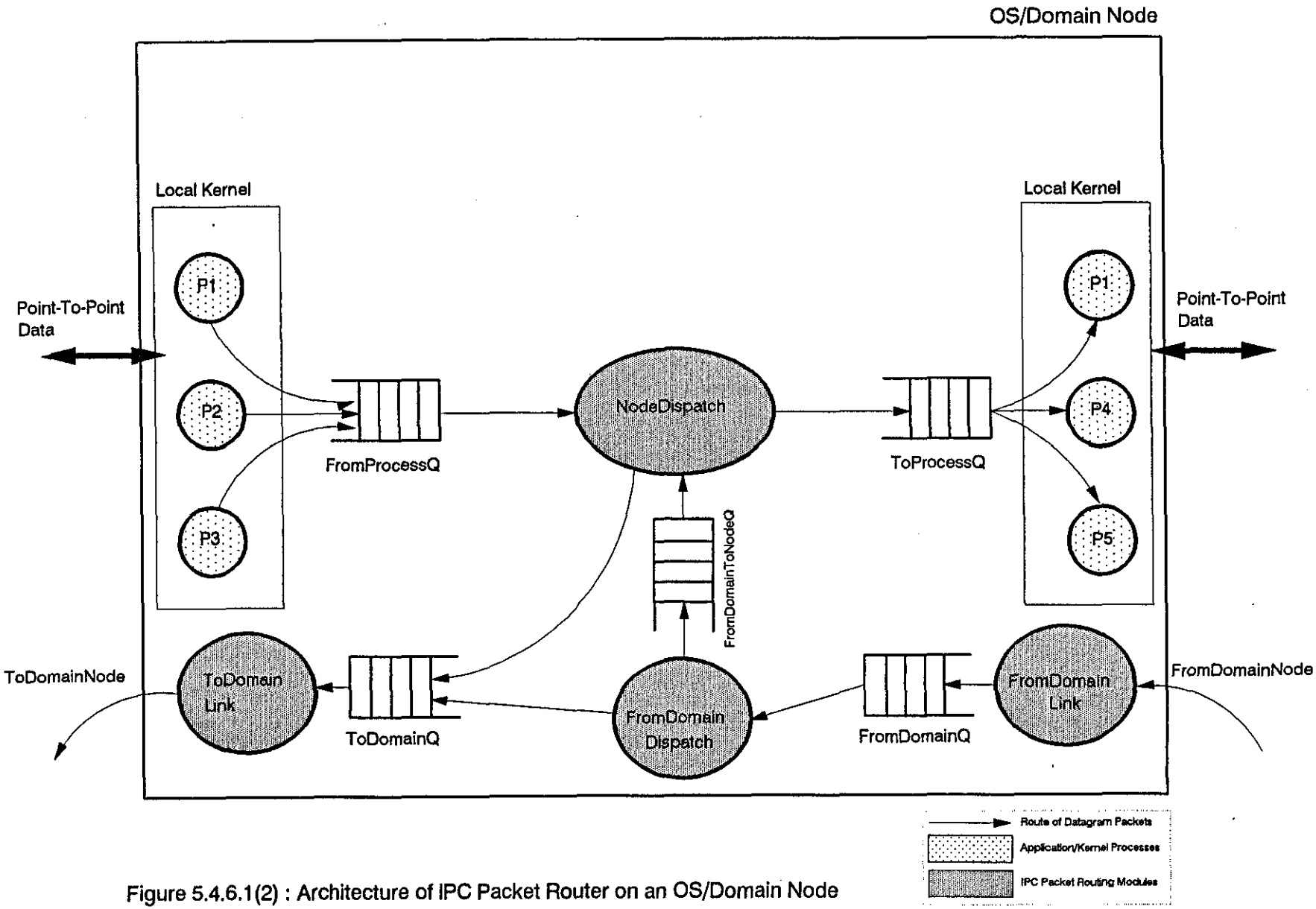


Figure 5.4.6.1(2) : Architecture of IPC Packet Router on an OS/Domain Node

A packet discarding mechanism is also implemented as part of the datagram delivery service. A packet deadline (last time to use) field is used to discard unuseful packets when they arrive at a node. A packet is discarded if its deadline time is earlier than the current global clock value. A discard bit is also used to eliminate packets which circulate a domain loop or *System Loop* once. No packet is allowed to visit any node twice in the network. This schema is useful especially to eliminate extra copies of broadcasting messages issued to locate process addresses. Regardless of the type of a node, all nodes execute these message discarding procedures, except that only the originator node of the message can use the packet discarding field to eliminate the cycling messages. The domain managers also act as originator node for the messages which are sent by some other domain but circulate in the current domain (broadcasts, multicasts, domaincasts, etc.).

A datagram packet can be fixed length (control commands used between kernels) or variable length (process messages). As is shown in **Figure 5.4.6.1(3)**, a packet consists of a header, source and destination addresses, and the message itself. A packet can be a minimum 36 bytes and a maximum 2039 bytes.

Packet Header consists of the following fields:

a) *Datagram Command* : One byte. This field is used to carry commands between kernels to set connections or notify them of important events. For process messages, it indicates if the packet is carrying data or not (99=packet carries data). A complete list of datagram commands are given in **Figure 5.4.6.1(4)**. The first eight of commands (01 to 08) are related to point-to-point connection establishment. Commands can be related to one destination (*Unicast*), or multidestination (*Broadcast*, *Multicast*, *Domaincast*, or *Domainmulticast*). Three of the commands are related to locating process addresses [TayliE 1990, Enslow 1978, StankovicA 1984]. The ultimate one is used as an acknowledgment, to inform processes which are waiting for a reply from the destination process (to establish reliable communication channels).

   In **Figure 5.4.6.1(4)**, possible originators and recipients of datagram commands are also given with the related reply command, if any.

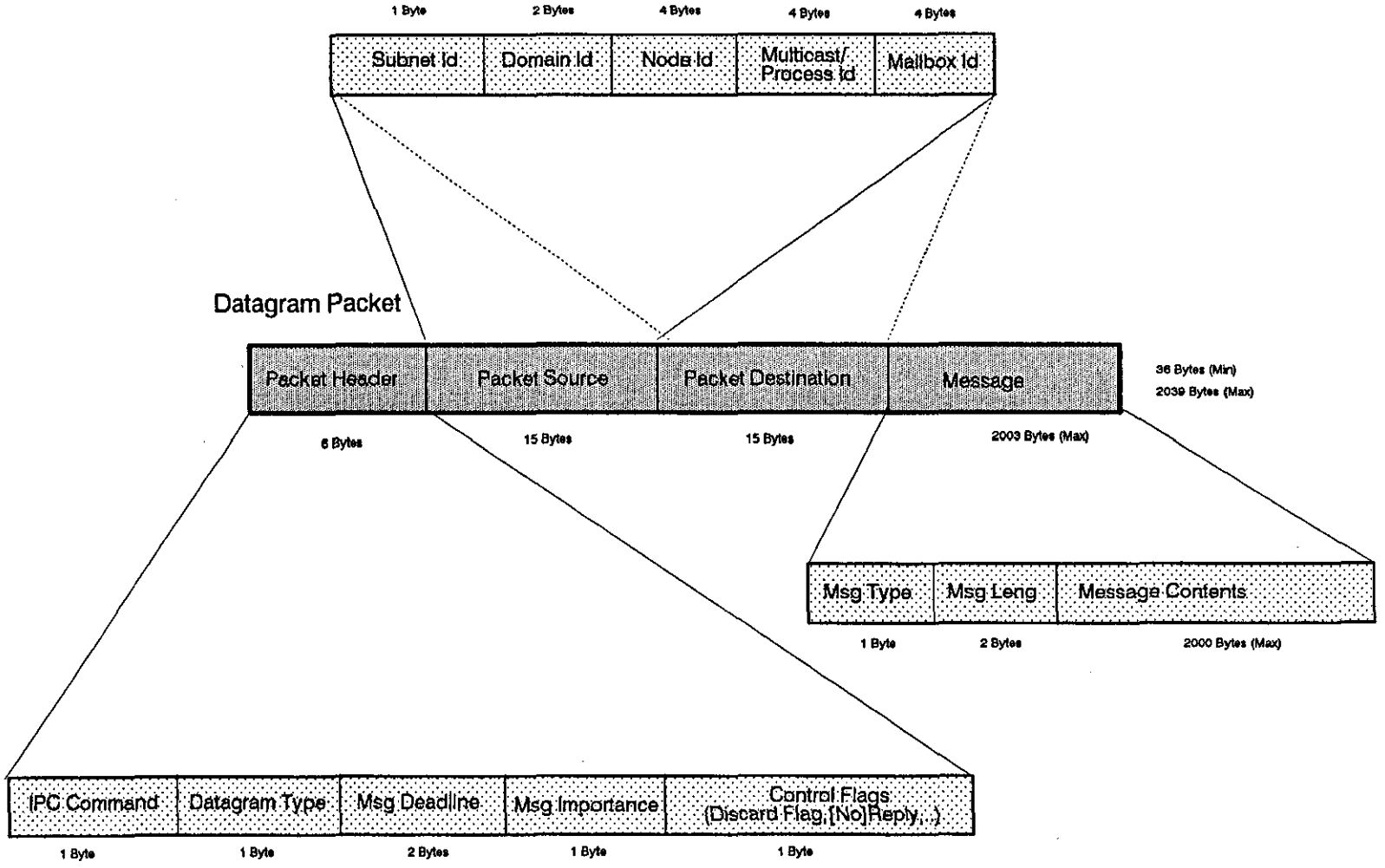


Figure 5.4.6.1(3) : Structure of an RT-DOS IPC Datagram Packet

Datagram Type	*IPC Related Datagram Commands*	*Originators*	*Targets*	*Possible Reply*
U	01 : Communication Request	Sender Process's Kernel	Receiver Process's Kernel	02
U	02 : Ready To Communicate	Receiver Process's Kernel	Sender Process's Kernel	-
U	03 : Connection Request	Sender/Receiver P. Kernel	Connection Manager	04
U	04 : Connection Ready	Connection Manager	Sender/Receiver P. Kernel	-
U	05 : Connect Transputer Links	Connection Manager	Switch Manager	06
U	06 : Transputer Links Connected	Switch Manager	Connection Manager	-
U	07 : Release Connection	Sender Process's Kernel	Connection Manager	08
U	08 : Disconnect Transputer Links	Connection Manager	Switch Manager	-
B,D	09 : Locate Process Address	Sender Process's Kernel	All Kernels	10
U	10 : Process Location Found	Receiver Process's Kernel	Sender Process's Kernel	-
U	11 : Process Address Changed	Receiver's Ex-Kernel	Sender Process's Kernel	-
U	98 : Acknowledgement	Receiver Process	Sender Process	-
U,B,M,D, DM	99 : Data Packed	Sender Process	Receiver Processes	98

U : Unicast
B : Broadcast
D : DomainCast
M : Multicast
DM : Domain Multicast

Figure 5.4.6.1(4) : RT-DOS IPC Related Datagram Commands and Possible Replies

For example, *Communication Request* command (01) is issued by sender process's kernel to the recipient's kernel for initiating a circuit switch between two processes. If the recipient process is ready for communication, its kernel replies with a *Ready To Communicate* command (02). If the datagram command is *Data Packet* (99), recipients can be a single process or a group of processes depending on the type of the datagram (for *Unicast*: one process; for *Broadcast*, *Multicast*, etc.: many processes);

b) *Datagram Type* : One byte. Datagram type is related to the number of destination objects, such as (U)nicast (to a single process), (B)roadcast (to all kernels), (D)omaincast (to all processes in an application domain), (M)ulticast (to a group of processes on the whole network), and (D)omain (M)ulticast (to a group of processes in a domain). Datagram type information is used for packet routing, and maps process level group communication requests directly to the datagram level primitives;

c) *Msg Deadline* : Two bytes. Message deadline is used to implement real time feature of the IPC. Value of this field is used to decide if message is still useful at any time before it arrives at its destination. If value of this field is earlier than the value of global system clock at any time, then packet is discarded assuming that it is useless for the destination process. Its value is derived from the related process parameter representing the user request;

d) *Msg Importance* : One byte. The message importance field is related to the real time feature of the IPC and will be used by upper level services. Its value (0 to 9) is an indication of the relative importance of the packet. It inherits the importance of the message specified by sender process which reflects the related parameter of the user request. For urgent messages its value is high so that the packet can be treated differently from ordinary packets. Its value can be increased if the deadline of a packet is very close, so that it can be treated as an urgent packet and arrives at its destination before meeting its deadline. Determination and semantics of its value are assumed to be handled at the process level and the IPC uses the value in routing of the packet through the network;

e) *Control Flags* : One byte. Each bit of the field is used for different control purposes. Currently it is only used for packet discarding purpose. Though packet deadline can be used to eliminate unused packets on the network, a discarding bit field can reduce the unnecessary cost of packet routing if it becomes useless before its deadline is reached. For example, if a packet circulates once through a domain loop and did not find its destination node on the domain, it should be discarded as a further circulation is useless on the same domain. The *Domain Manager* sets the field the first time the packet enters the domain, and eliminates it when it circulates all the nodes on the domain and arrives at the same node again, by checking this field value. If it is set when it arrives at the domain manager node, it means that this is the second entry of the packet to the node. If the source node identification of any packet is the same as the current node identification when it arrives at the node, it means that the packet has circulated the current loop at least once and it should be discarded.

Packet Source and *Packet Destination* have the same fields and are used for packet routing purposes. Each packet carries source and destination address information, but in some exceptional situations the destination address may not be required. Though carrying routing information (full address) with the packet each time incurs some extra communication cost, it offers the flexibility of changing process locations dynamically and reduces error recovery costs. Some fields within the "*packet destination*" segment of the message will be empty, if the destination object's location is not known when the packet is issued. The fields and their values are as follows:

a) *Subnet Id* : One byte. The subnet identification is used to indicate source or destination network, if a number of networks are connected to each other via gateways. If the current network identification is different from the destination network identification, then the packet is directed to the destination network via the relevant gateway node. The current implementation doesn't assume the existence of a multinetwork topology. The *Subnet Id* is a system wide unique number;

b) *Domain Id* : Two bytes. Domain identification is used to identify source or destination domain of packets. If the packet destination is not known, or the datagram is related to a group communication protocol, such as multicasting or

broadcasting, then this field is invalid (null). If the destination domain of a packet is different from the current domain, the *Domain Manager* sends the packet to the *System Loop* only. If the destination domain is the same as the current one, the *Domain Manager* sends the packet to the next domain node (if the destination node is not the application domain node itself), instead of the *System Loop*. The "*packet source*" carries the domain information all the time, so that the receiver's reply can be sent directly to the source address. It is a system wide unique number, and dynamically generated by a centralized name server;

c) *Node Id* : Four bytes. It identifies the source or destination process's node address. It is used for packet routing purposes. The "*node identification*"s are globally unique and randomly generated from a 32 bit wide address space by a centralized name server. Each packet sender includes this field as part of the source address. If the packet sent is a multidestination message or the sender doesn't know the receiver's node address, it is not included in the packet destination "*Node Id*" field;

d) *Multicast/Process Id* : Four bytes. This field is included in the "*packet source*" as the process identification at all times. For the "*packet destination*", it can be either a multicast identification (if the packet type is *multicast* or *domain multicast*) or a process identification (if packet type is *unicast* and the destination process is known). The "*process*" and "*multicast identification*" are globally unique and randomly created by a centralized name server. This field is empty if the packet type is *broadcast* or *domaincast*;

e) *Mailbox Id* : Four bytes. The *mailbox identification* is included in the "*packet source*" segment of the messages at all times. In the "*packet destination*" segment, this field is empty for all group communication messages (*broadcasting, multicasting, domaincasting,* and *domain multicasting*). The mailboxes are created dynamically with node-wide unique identifications. To establish a connection between two processes a mailbox is created by the relevant kernel of each process for exchange of protocol packets. Each communication entity on a node has a mailbox for every connection with other processes over the network. A kernel has one port (but as many mailboxes as

interprocess connections) for each process on the network. A connection is uniquely identified globally by a combination of two corresponding mailboxes and process identifications.

The "*message*" segment of the packet is included in the datagrams if the "*message type*" field is "*Data Packet*" (99), and it is not present (ignored) for the IPC command packets which carry no user messages. It contains three fields :

a) *Msg Type* : One byte. It is reserved for usage in the implementation of the process level end-to-end protocols. For example, integer and real type data representation on 16 bit and 32 bit Transputers are different and a data conversion procedure is necessary on heterogeneous networks. The sender's kernel updates this field to notify the receiver's kernel about the type of data sent. The semantics of usage of this field is out of the scope of the thesis. This field is ignored if the datagram type is not "*Data Packet*" (99);

b) *Msg Leng* : Two bytes. The length of the message is filled by the sender's kernel during the packet assembly process considering the length information of the user message. This field is ignored if the "*datagram type*" is not "*Data Packet*" (99). A message length carried by datagram packets can be a maximum of 2000 bytes. The reason for this limitation is the assumption that the cost of a point-to-point connection establishment between two processes is about the same as transferring 2000 bytes (as it is shown later on) through network nodes from source to destination using datagram service. If the length of user message is more than 2000 bytes, establishing a physical connection between source and destination Transputers and transferring data through physical links is more convenient in terms of network communication costs. An elaboration of this issue is presented in **Chapter 6**;

c) *Message Contents* : 1-2000 bytes. Processes can exchange messages either by establishing direct physical connections between nodes on which they reside, or by sending the messages through the network using the datagram services. When to set direct connections or when to use the datagram service is a policy matter; and must be the decision taken by the higher layers. An upper layer requests the use of the most convenient data transfer method for its needs, considering the semantics of the communication it chooses. If the message length is less than

2000 bytes or communication type is not a kind of group communication, then using the datagram service directly is assumed to be more advantageous than setting a point-to-point connection between communicating entities.

5.4.6.2 Circuit Switching Service

The Transputer hardware allows direct point-to-point connection and communication between neighboring Transputers using high speed serial links. Classical architectures for Transputer-based systems are built around arrays or hyper cube interconnection topologies. With these types of topologies, any communication between non-neighboring nodes in the network typically involves the activity of many other nodes. This becomes very costly because of buffer allocation and transfer of data from node to node, if messages are of large size and/or frequent.

The multiloop topology of the application domain based RT-DOS architecture reduces the cost of communication by isolating the local traffic of each domain from the overall system. However, allowing the direct node-to-node connections between any two Transputers to support interprocess communication is a necessity in some conditions, especially for bulk data transfer between physically distant processes.

With the *C004* dynamic link switches, circuit switching mechanism is used to implement direct node-to-node connections. These circuits are established on request for the duration of one message exchange between a pair of processes and relinquished afterwards to allow another pair of processes to communicate.

From the IPC point of view, the *Datagram Layer* which has been built on top of the *Physical Layer* is considered as a communication back plane on which two communication models (circuit switching service and message switching services) are built.

The *circuit switching service* is implemented by exchanging seven datagram packets between communicating entities (local kernels of two communicating processes and connection service). As is seen in **Figure 5.4.6.2(1)**, one of the processes (the *initiator*) initiates the communication by sending a *Communication Request* to the second party (the *responder*).

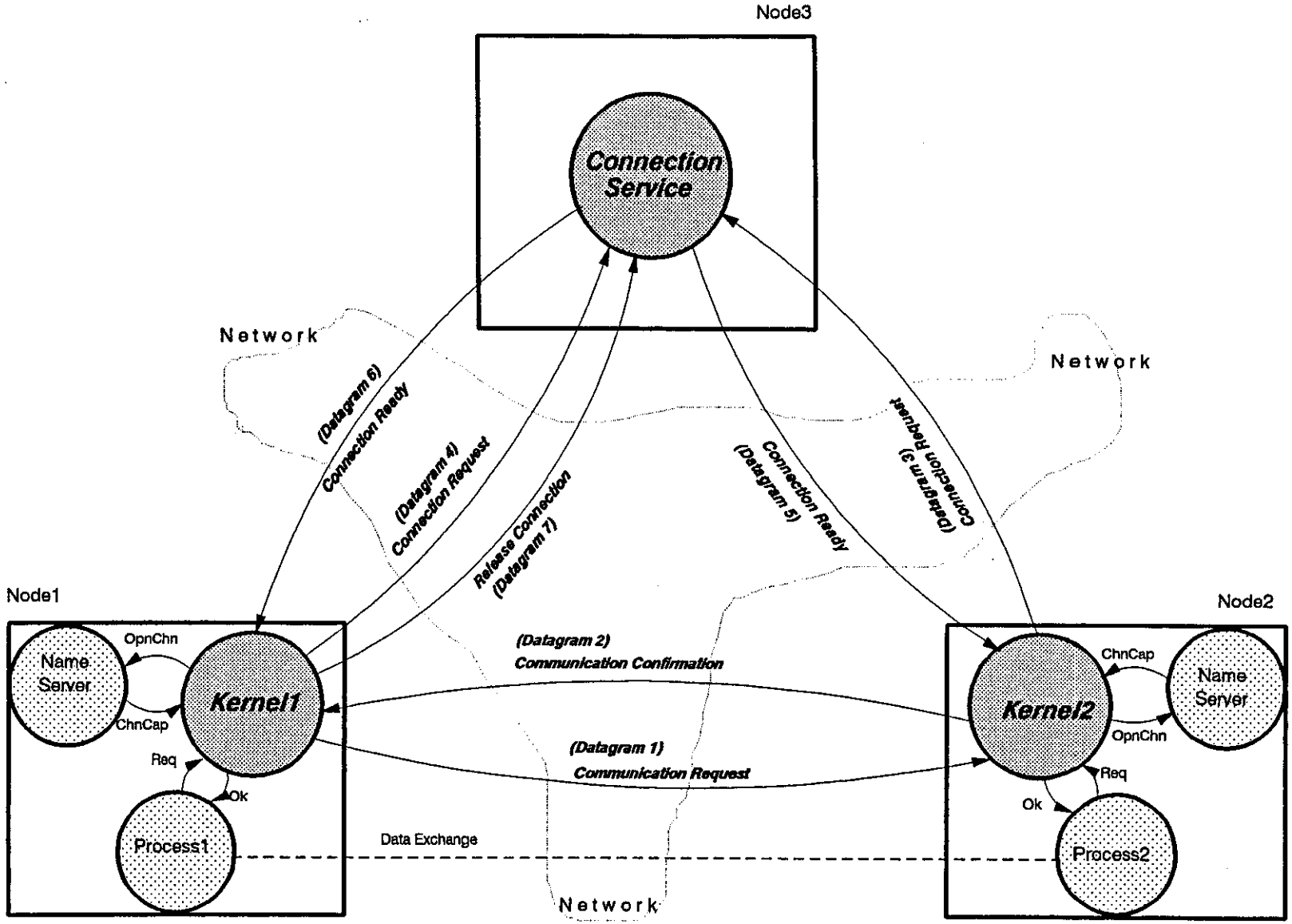


Figure 5.4.6.2(1) : Datagrams (7) Exchanged Between Processes to Set Connection

In this schema (the current implementation), the *initiator* acts as client while the *responder* acts as a server. The server can have connections with more than one client process, but it can not establish more than two (maximum) concurrent connections at any time, as a node has a maximum of two free links to be used for point-to-point relations. A client process can have only one connection with any other server processes at any time.

Consider a *file server* which is waiting for file manipulation requests coming from user processes. Each user process issuing a file access request (read or write) to the *file server* is acting as a client. The user process must initiate the communication, as the file server can't anticipate the intention of any other process on the network without getting a service request from them.

The server process can reject the communication request of the client by ignoring it if available resources (on the server node) to be used for connection establishment are already exhausted (buffers, links, etc.). If a server accepts the communication request of the client, then it sends a reply to the client's kernel (a "*Communication Confirmation*" datagram command). The client processes use the time-out technique to check the status of connection requests.

After accepting the communication request of the client, the responder's kernel sends a "*Connection Request*" command to the connection service which consists of a number of complementary service agents with different functionality. The initiator's kernel also sends a "*Connection Request*" datagram to the connection service after getting the "*Communication Confirmation*" command from the responder. Then, the connection service matches the connection requests of client and server processes.

After setting the connection through available links of the related nodes, the connection service informs both parties by sending a "*Connection Ready*" notification to them. After transfer of data through the connected Transputer links, the client site sends a "*Release Connection*" notification to the connection service, so that the unused (idle) Transputer links can be reused.

Figure 5.4.6.2(2) shows the connection establishment steps in their order. In the figure, *Process1* acts as client site while *Process2* acts as server site. The connection service is represented by three system server agents : the *Connection Manager*, the *Switch Manager*, and the *Switcher*. These connection service managers will be explained later on with their important data structures. *Kernel1* and *Kernel2* residing on the same nodes are *Process1* and *Process2*, respectively.

Process1 and *Process2* indicate their intention of communicating with each other by issuing input/output commands (*ChannelIn* and *ChannelOut* with the *Prm1* parameter). The *Prm1* parameter consists of the destination process identification, message buffer, message length, time-out (deadline), and message importance. The *message importance* is normally inherited from the priority of process which issues the I/O command, unless it is explicitly specified differently with a parameter.

Message time-out is used to avoid deadlock or starvation conditions, and to eliminate the unwanted messages. After agreeing on the communication requests, the kernels of the client and the server processes send connection requests to the *Connection Manager*. The *Connection Manager* then matches connection requests by using the channel capability sent by both parties. The channel capability consists of the client and the server identifications and the message mailboxes created during the communication exchange before the connection requests.

To avoid deadlock or starvation conditions, the *Connection Manager* eliminates the unmatched connection requests after a time-out. Established communication channels are also released automatically considering their ages if the "*Connection Release*" notification of the client doesn't reach the *Connection Manager* in time, for any reason. After matching the connection requests, the *Connection Manager* sends a "*connection order*" to the related queue of the *Switch Manager* with the *Prm2* parameter. The *Prm2* parameter consists of the identifications of nodes to be connected and the channel capability of the connection to be established. The *Switch Manager* checks available links on the given nodes. The *Prm3* parameter consists of nodes on which status of links are checked. If both nodes have free links, then the *Switch Manager* sends a "*Connect Order*" to the *Switcher* process which is responsible for configuring the *C004* programmable link switches.

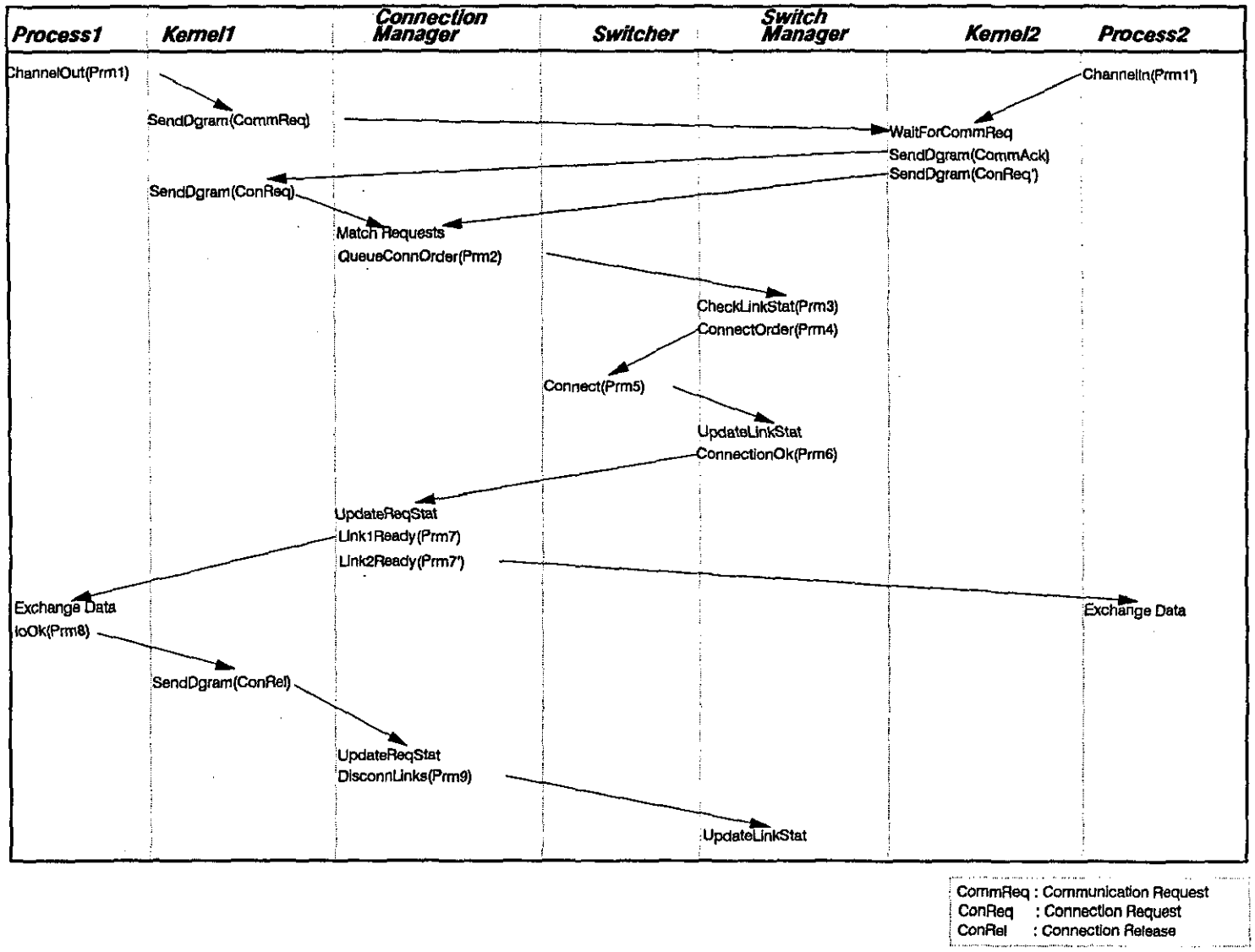


Figure 5.4.6.2(2) : RT-DOS IPC Connection Establishment Protocol

The *Prm4* parameter contains the identifications of both nodes and links to be connected. The *Switcher* maps the Transputer link numbers to the *C004* switchboard link numbers. The *Prm5* parameter contains the *C004* link numbers to be connected. After connecting the Transputer links, the *Switcher* informs the *Switch Manager* about the connection result, so that it can update the status of the links accordingly. A busy link can't be used for any other connection until it is released. After that, the *Switch Manager* updates the status of the links and informs the *Connection Manager* with the parameter *Prm6.* The *Prm6* parameter consists of the node identifications, connection capabilities, and link numbers. After updating the connection status, the *Connection Manager* notifies both communicating parties about the status of their connection requests. If the connection establishment is successful, then the communicating processes start data transfer through the physical links which are defined in the *Prm7* parameter. If the processes requesting the communication can't get a notification before the time-out period, they assume that the connection request is not successful. In such a case, the method for handling the situation (retry or abort) is up to the application layer (policy) protocols, rather than the IPC itself. After exchanging data successfully, the client process's kernel notifies the *Connection Manager,* with a "*Release Connection*" command, of the completion of the communication, so that related links are released to be used in other connections. The *Connection Manager* sends Transputer numbers and link numbers with the *Prm9* parameter to the *Switch Manager*, and the *Switch Manager* updates the connection status of the related *C004* links by mapping Transputer link numbers to them, using the configuration tables. In fact the links are not physically disconnected from each other, as there is no need for such an overhead. Instead, the status of the links are updated in the configuration tables (i.e., changed from "*busy*" to "*idle*").

Figure 5.4.6.2(3) shows the basic components involved in a connection establishment protocol, and the steps followed in the process. **Figure 5.4.6.2(4)** also illustrates the implementation of the data structures of the *Connection Service*. The *Connection Service* consists of two service managers (the *Connection Manager* and the *Switch Manager*) and a *Switcher* agent process. Connection service managers can be physically on the same Transputer (node) or different Transputers. They exchange the circuit switching protocol commands through the raw datagram service.

The *Connection Manager* maintains a *Request Matching Table* to match connection requests of communicating parties. After matching the requests of two related processes,

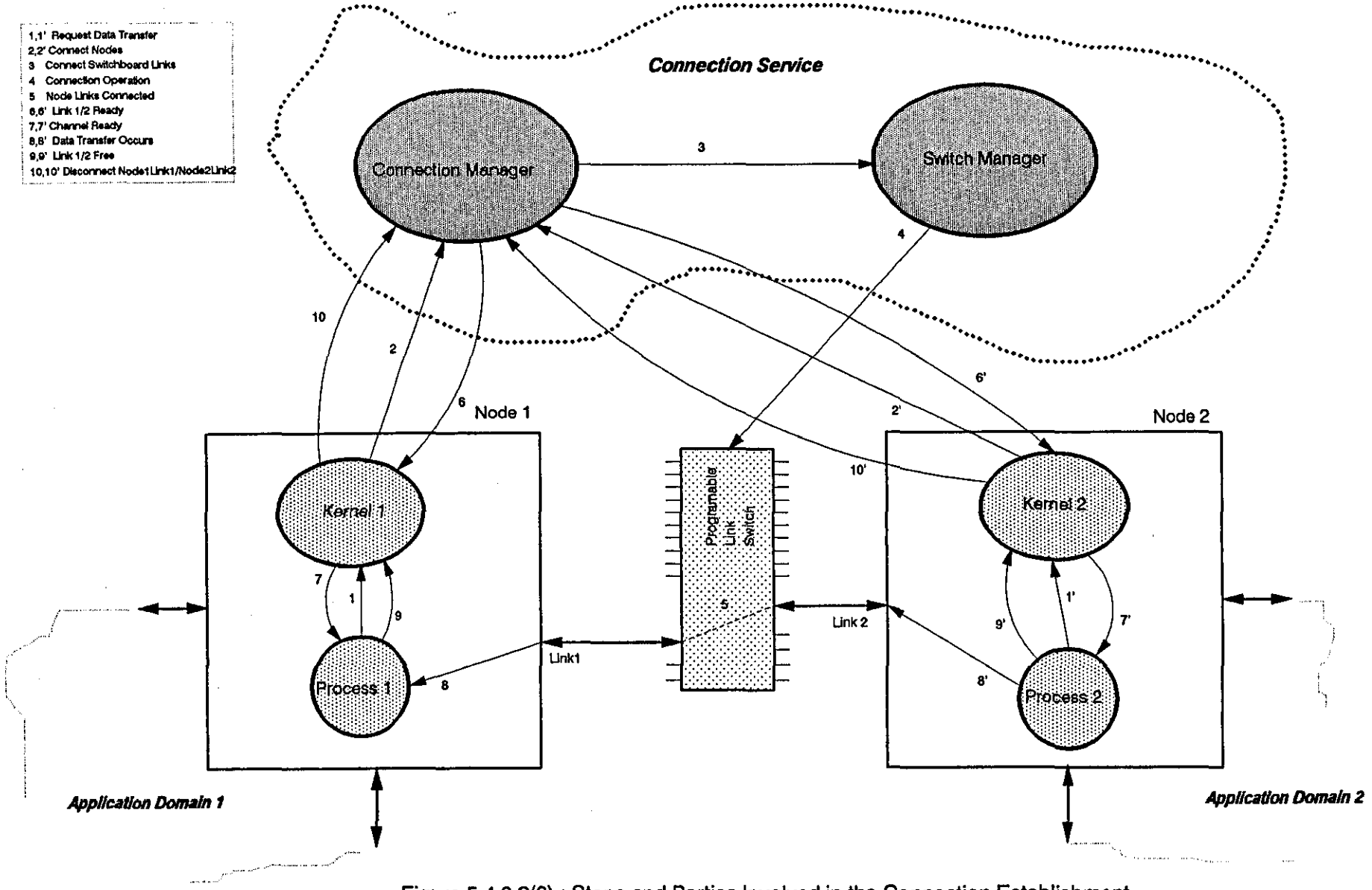


Figure 5.4.6.2(3) : Steps and Parties Involved in the Connection Establishment

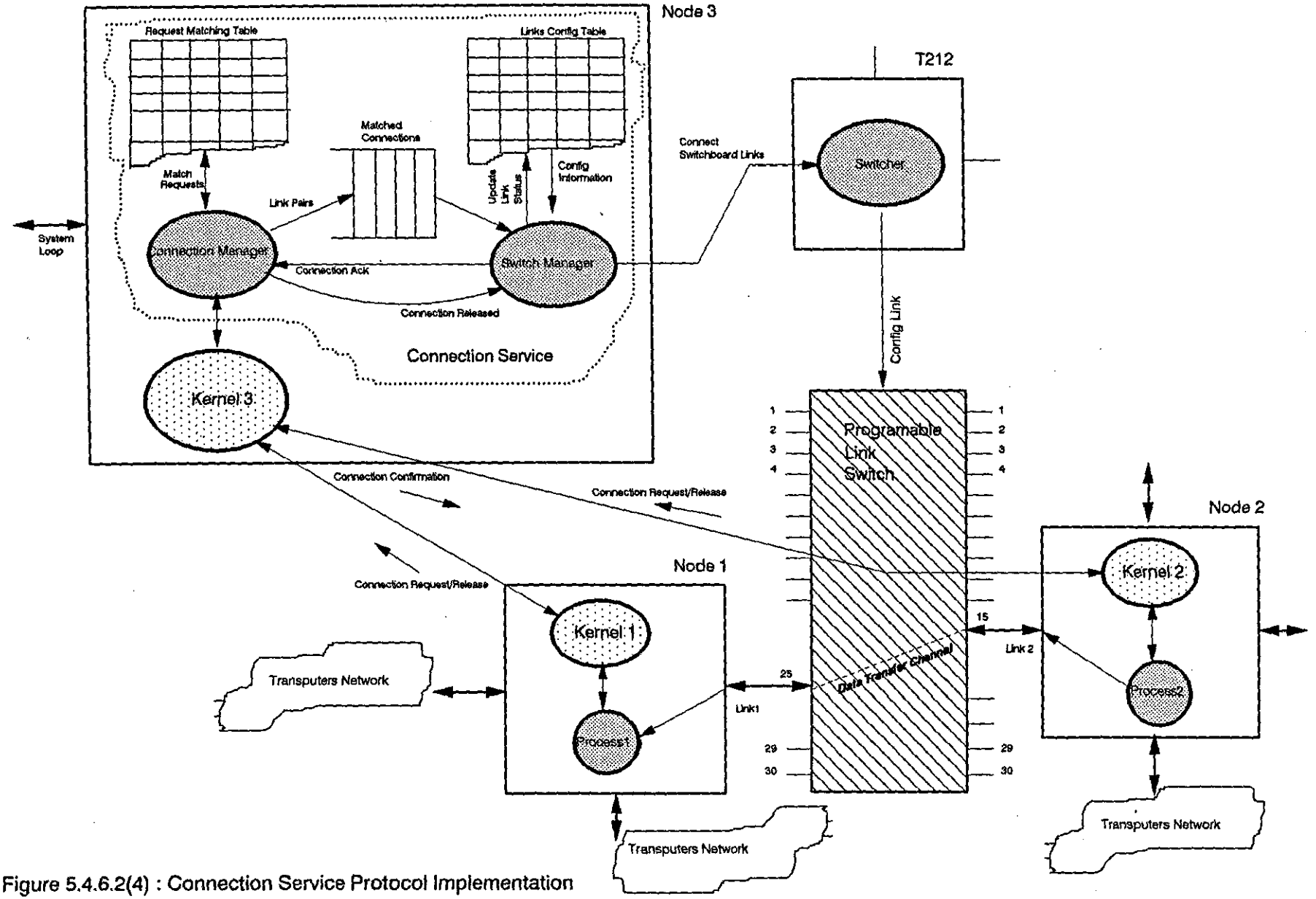


Figure 5.4.6.2(4) : Connection Service Protocol Implementation

it sends node pairs to be connected to the *Switch Manager*, through a queue. For any new connection request, the *Switch Manager* checks the *Request Matching Table*, and if it finds a matching request in the table it queues the matching requests, so that the *Switch Manager* connects related nodes when it finds two free links on the nodes. If there is no matching request inserted in the table in advance, the request is inserted in the table and kept until the time-out period or a matching request arrives from the other party. Each entry of the *Request Matching Table* has initiator's identification and physical location (node and domain address), responder's identification and its physical location, and connection identification (combination of process identifications and mailboxes). The *Connection Manager* waits until it gets a confirmation (from the *Switch Manager*) about the success of connecting two nodes through two free links. It sends a notification to the communicating processes after getting this confirmation.

The *Switch Manager* gets matched connection request from the *Connection Manager* through the *Matched Connections* queue (node pairs to be connected) and checks the nodes for available links to be connected. It maintains a *Link Configuration Table* which keeps tracks of status of all links of Transputers. Some of the links are dedicated for permanent connections between two nodes. The table is created (loaded) at the beginning of system initialization or reconfiguration, with all available nodes, links, and their connectivity. Mapping of Transputer link numbers to the *C004* switchboard links also is carried out at this stage. After connecting any link, the *Switch Manager* updates its status (from *"idle"* to *"busy"*); and sends a positive acknowledgment to the *Connection Manager*.

After mapping Transputer and link numbers to the *C004* switch link numbers, the *Switch Manager* orders the *Switcher* to configure switch links accordingly, by sending the *C004* configuration commands. The *Switcher* process is residing on a different node (a T212 Transputer reserved for configuring the *C004* switch links). T212 Transputers, dedicated for the *C004* link configuration, has the only 2KB of on-chip RAM. The Inmos Parallel ANSI C compiler, which is currently used for the IPC implementation, can not generate code less than 4KB. Because of this specific technical reason, the *Switcher* process is coded in the OCCAM language, compiled with the TDS compiler which can generate code with size less than 2KB and loaded on this Transputer with a specific loader during initial configuration. It accepts the *C004* link configuration commands and downloads this command to the *C004* programmable link switch through its *Config Link*.

For the details of *C004* dynamic link switch and its configuration commands, please refer to **Section 5.4.3**.

5.4.6.3 Packet Switching Service

For bulk transfer of data (more than 2000 bytes) between processes, the point-to-point connection is advantageous; but for small messages, the cost of setting a connection (number of datagram packets exchanged between communicating parties), in terms of network traffic and number of node interrupts, is far more than sending the messages directly to the destination through the datagram service. In addition to this, the processes on the domain manager nodes can't set any point-to-point connection with any distant process as these nodes don't have any extra link to be used for a connection establishment. The message switching service is the only means of communication for them. Also for the group communications, the point-to-point communication service is very expensive comparing to the distribution of the messages through the datagram service, directly.

Because of all these reasons, the message switching service is provided as an alternative communication model to the circuit switching technique. The message switching protocol is implemented at the datagram service level by mapping the data transfer request of a process to a single datagram command. If the datagram type is "*Data Packet*" (99), then the user message itself is forwarded from node to node with the datagram packet until it reaches its destination. The sender process provides the datagram service with the destination process identification, message length, message itself, a time-out (0=nowait or deadline), importance of the message (0 (unimportant) to 9 (urgent)), and a mask indicating whether a reply is required (for reliable communication) or not (for unreliable communication). All those parameters are represented within a field of the datagram packet format and each datagram packet carries this information to the destination.

During the packet assembly process, the sender's kernel checks the local address cache buffer if the destination address (domain and node identifications) is known or not, using the globally unique process identification. If it is found, the message is sent directly to the destination after assembling the message with a datagram packet.

If the location address of the destination process is not known a priori, the sender's kernel broadcasts a "*Locate Address*" command, and receives a reply from the receiver's kernel with the node and the destination address of the recipient. The sender's kernel registers the distant process's address in the local cache and sends the message to the destination. If the sender expects a reply from the recipient, it waits until time-out. The sender's kernel notifies the sender process with a status which indicates either a failure or success of the last communication attempt. If the sender doesn't receive a positive acknowledgment in the time-out period it assumes that the message did not reach the destination. How to handle this erroneous situation is up to the high level policy layers.

The receiver process opens a mailbox and waits for the sender's message until time-out. The semantics of the message is defined between the communicating processes (end-to-end protocol).

The IPC routing service forwards the message bundled in a datagram packet, from one node to another node until it reaches its predefined destination. Packet discarding is carried out by the datagram service. If the message deadline is reached before the packet completes its circulation all through the network, then it is discarded automatically. The sender process is not notified of any network or communication failure. Only in the situation when the receiver's address has been changed, is the sender's kernel informed, by a "*Process Address Changed*" (11) command, so that it can update its address cache accordingly. The kernel resends the message again if it is required, considering the semantics of the protocol defined between communicating parties. The sender should broadcast a "*Locate Address*" command to locate the new address of the receiver if it wishes to repeat the message transfer. The kernel of the former receiver process does not keep track of the new address of the receiver.

Though this approach is inefficient and increases the network communication costs, it provides flexibility and location independence for process migration and reduces the error recovery costs. Because of the efficiency considerations, the sender process can be informed about the new location of the receiver process, by maintaining the new address in the address cache of the former kernel.

Congestion is also handled by the datagram delivery service. If the resources of any kernel are insufficient to handle the last packet arrived, just because of the lack of the

message buffers or for any other reasons, the message is discarded (ignored) immediately, without notifying the message sender. The message sender waits until the time-out and assumes that the message is lost or rejected, if the communication status indicates a failure (if sender expects a reply for its message). Error recovery, in a deadlock or starvation situation, is handled by end-to-end process level protocols defined between communicating parties.

5.4.6.4 Group Communication Services

Most, if not all, of the systems are based on some form of broadcast communication media (e.g. ring or Ethernet hardware) and so the lower levels of the IPC can take advantage of this and ignore routing problems. Although this is not the case with current Transputer based systems, the RT-DOS multiloop topology does provide a useful starting point to implement an efficient infrastructure for establishing group communication protocols.

There are four types of transport level group communication protocols implemented at the IPC level of the RT-DOS. These are *Broadcasting*, *Multicasting*, *Domaincasting*, and *Domain Multicasting*. Each one of them implements a different type of *N:1* communication (a number of clients access only one server) at different context.

Broadcasting is used to locate address of a process on network. It is implemented at the IPC datagram service by mapping to a datagram command. Destination fields of the packet are empty, and the packet visits all nodes all over the network even if it arrives at the intended destination process. The packet is duplicated at each domain manager node by the packet routing service and sent to the each domain loop, as well as the next node of the *System Loop* on its way. Each duplicated packet is discarded either after it circulates the loop once, or at the destination process after it reaches there, or anywhere on the network if the packet deadline is reached, or at the source node if it returns back after circulating the related loop once. If the sender gets more than one reply for a *Locate Address* command, it accepts only the first one and ignores the rest.

If the datagram type is *Broadcast* and the IPC command is defined as *Data Packet,* then the message is distributed to all kernels on the network. Because of the efficient routing algorithm supported by the RT-DOS multi loop topology, a broadcasting message circulates all network without passing through any node more than once.

The second group communication protocol is the *Multicasting* protocol. It is used for sending a message to a group of processes distributed all over the network. *Multicasting* is also implemented with a datagram command at datagram level. The datagram type is *Multicast* and the IPC command field is *Data Packet*. The destination domain, node, and mailbox identifications of the datagram packet are empty, while process identification is interpreted as multicast identification. Each process has a globally unique multicast identification (group membership identification) as well as unique process identification. Uniqueness of multicast identification is maintained by a centralized naming service, in the same way as process identifications. Semantics of group identification, the decision to add a member (process) to a group, canceling group membership of a process, etc., are out of the scope of the IPC implementation. These issues are handled by process level policy layers. The IPC datagram delivery service can deliver a packet to all members of a group regardless of their physical location at the moment of packet distribution.

Multicasting is treated as broadcasting by the packet routing service. Each process's multicast identification on each node of all application domains and system domain loop are checked for the similarity of destination multicast identification. If multicast identification of any process is the same as the destination multicast identification of the packet, then the packet (message) is copied to the mailbox of the process. Duplication of the packet through domains, discarding of unused or aged (or repeating the same route) packets, and other routing problems are handled in the same way as a packet broadcasting.

The IPC level support for multicasting service can be useful especially for multiple copy update problem. If any message (a notification or statistical information) should be displayed on all terminals, then a copy of the message is sent to each terminal server.

Multicasting has been recognized as a powerful facility in distributed systems for implementing decentralized naming, distributed scheduling, parallel computation, distributed transaction management and replication [Kaashoek 1990].

The third IPC group communication protocol is *Domaincasting*. *Domaincasting* is a broadcasting restricted to one application domain only. It is also implemented by a single

datagram command. Packet destination fields are empty except the *Domain Id* field, like in broadcasting, and the packet type is *(D)omaincast*. It is useful for informing all processes of an application domain about an event, or distributing a message to the whole domain context without affecting network traffic performance at other domains.

The last IPC group communication protocol is *Domain Multicasting* which is a message multicasting restricted to a single domain. All multicasting group members which receive the message will reside in the same domain space. The datagram type is *(D)omain (M)ulticast* and packet destination fields are empty, except *Domain Id* and *Multicast Id*. It can be used to handle multiple copy update problem in the domain context. For example, if a replicated copy of a file is maintained with its original copy to increase fault tolerance, this IPC group communication service can be used to implement this protocol. It is implemented at the IPC level by a single datagram command. The datagram routing service treats the domain multicast packets as combination of domaincast and multicast. A message is copied to all group members residing in the same domain.

5.4.7 The IPC Data Structures and the Message Layouts

During initial loading and system reconfiguration each node is given a unique identification and some data structures are initialized to support the IPC packet routing and delivery services. Each node is given a network identification (*MyNetwork*), domain identification (*MyDomain*), and node identification (*MyNode*). Connectivity of each link (permanent Transputer-to-Transputer connections, *C004-to-Transputer* link connections, etc.) on a node are also defined in the *Switch Manager*'s link configuration table (*Links Config Table*).

In **Figure 5.4.7(1)**, the basic IPC related node data structures are given. *DomainInLink* and *DomainOutLink* are physical link numbers which connect the current node (Transputer) to the previous and next nodes of the current domain loop. *OsInLink* and *OsOutLink* are physical link numbers of the current node which connect it to the previous and the next nodes of the *System Loop*, respectively.

These links are free on nodes which are not domain managers, and used for point-to-point connection establishment. *NodeType* indicates if the node is an ordinary node on any loop (a domain or the *System Loop*), or it is an application domain manager node as

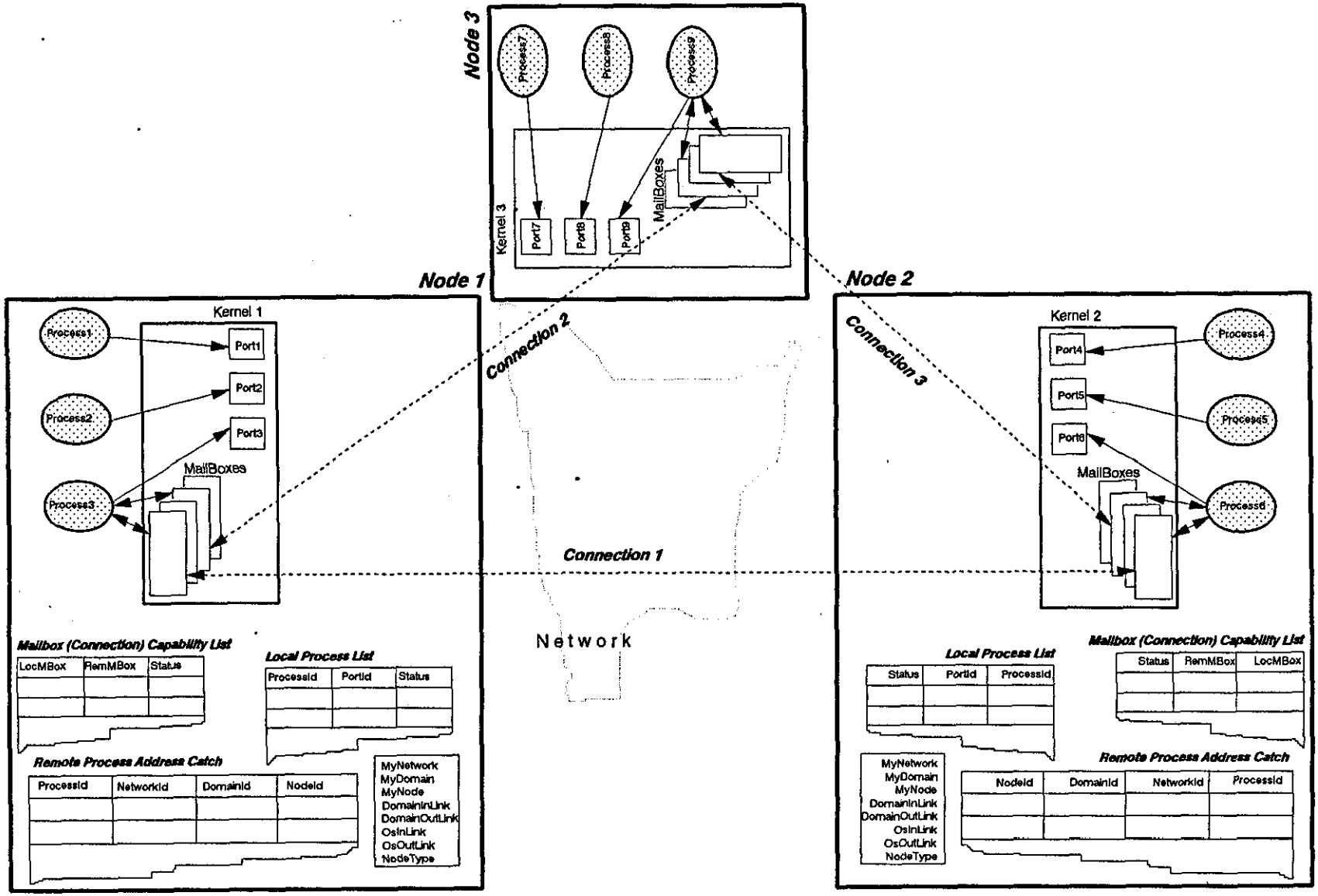


Figure 5.4.7(1) : Basic IPC Data Structures of an RT-DOS Node

has been shown. The datagram routing algorithm, packet discarding policy, error recovery and some the IPC related services act differently on domain manager nodes, though the same copy of the IPC is running on each node.

As an example, the domain manager nodes don't have any extra links for point-to-point data transfer, and any process on these nodes must use message switching service when communicating with any other distant processes. Broadcast and multicast messages are duplicated on these nodes and sent to domain loops as well as the next node on the current loop.

There are two basic tables on each node regardless of the their type, in addition to specific tables and data structures related to servers which reside on the node. The *Local Process List* table is maintained to keep track of active processes on each node. Their globally unique identifications, symbolic names, multicast identifications, port identifications and the list of active mailboxes related to the connections with the other processes, are the basic important fields which are directly used by the IPC datagram routing service.

The *Remote Process Address Cache* table is maintained for efficiency reasons. Addresses of the latest referred remote objects (processes, servers, etc.) are kept in this table to increase performance of the IPC datagram service. If the physical location of any process is known before it is sent a message, it is not necessary to broadcast a *Locate Address* datagram command to find the address of a process. For each process referenced, the table has an entry which consists of process identification, network address of the process, domain identification, and node identification.

For each circuit switching or message switching session, the kernel creates a mailbox on each nodes involved in a connection, to keep track of the latest status of communication and the exchanged messages.

5.4.7.1 Message Buffers

Communication bandwidth, node buffer space and node processing power have been the critical resources for message passing systems. The advantage of using a fixed packet size are twofold: it enables fast packet processing at the physical layer, as buffers could be statically allocated, and as such it increases the overall network bandwidth; it may also result in a more predictable (deterministic) network response, as individual clients can't lock a given link with the transmission of extended packets. However, the implementation of a fixed packet size would significantly reduce the end-to-end bandwidth, as it necessitates an additional copy of all message data from user buffers to kernel (physical-layer) packets at both ends of the transfer.

The RT-DOS IPC uses fixed size datagram packets routed from node to node using a store and forward technique, without error checking and hence datagrams may be corrupted or lost (reliability is assumed to be implemented by higher level layers, including message sequencing and error recovery). For all processes on each node, a message buffer is created for both sending and receiving messages. These message buffers are called mailboxes, and for each connection between two processes a mailbox is created at both sites to identify the connection uniquely system wide. Mailbox identifications are unique on each node and maintained by the local kernel. For any process on a node, there might be more than one mailbox at any time, especially if the process is a server. For example, a file server might have a connection with each one of a number of clients at any time, concurrently. There is no limitation about the number of connections with a server and its client processes, except the availability of local resources such as memory. For each connection request coming from a client process, a message buffer (mailbox) is allocated dynamically if the request is accepted. This schema supports N:1 mode communication. For any client process, it is assumed that there might be more than one active mailbox as well; assuming that the granularity of computing elements of the RT-DOS kernel is at the thread level (subprocesses running concurrently) rather than processes themselves.

For each mailbox message length, message buffer, message status, local process identification and remote process mailbox identification relating to the active connection are maintained. A stub process to assemble and disassemble messages coming/going to/from a mailbox is created by the kernel when the mailbox itself created.

5.4.7.2 Communication Ports

Communication entities that wish to establish a connection first create a mailbox identified by a capability representing the requested connection or communication channel. The mailbox is attached to a local communication port and initialized with the proper unmarshalling procedure. Subsequent connection requests refer to the mailbox. The triplet channel identifier, port location, and port identifier names a connection request uniquely under any condition. The identification of sender and receiver is combination of domain identification, node identification, and port number. Ports are seen as mailboxes internal to a kernel, and allow non-ambiguous delivery of datagrams. Basic system servers (such as the *Connection Server* and the *Name Server*) have publicly known port names. Each process has a port (but a number of mailboxes) related to it and created by the local kernel to managing packet distribution. A packet directed to a process is put first to the local port of the process and then copied to the related mailbox.

In the following chapter (**Chapter 6**), some implementation problems and the implications of a number of different alternative solutions for these problems on the RT-DOS performance are elaborated. Some of these claims are tested and reconfirmed after the real implenetation, at the end of **Chapter 7**.

6 ELABORATION OF IMPLEMENTATION ALTERNATIVES AND SOME IMPROVEMENT AREAS

The biggest bottleneck of the current RT-DOS IPC implementation is availability of system resources for point-to-point connection establishment. These are the centralized connection service, network bandwidth to carry datagram commands between communicating parties (connection request, release, locating address of destination, etc.), and processing power of network nodes to handle interrupts related to datagrams circulating all over the network.

One alternative to reduce the number of datagrams to be exchanged between communicating entities to establish a connection is to send requests directly through physical links dedicated for this purpose, instead of broadcasting them through the network. **Figure 6(1)** presents this approach quite clearly. In this approach, one of the four links of each Transputer is dedicated for only sending connection requests and getting replies from connection service (*Fixed Connection/Disconnection Request Links*). A polling process as part of the connection service can regularly poll the links by connecting the special link of config Transputer on which it resides to the each link of application Transputers in turn. If a process on any of these Transputers requests a connection with a remote process, it will forward its packet through the dedicated link to the polling service when it is polled. Another process of the connection service (the *Matching Process*) can match all collected requests as currently required and then forward to the *Switch Manager* which, in turn, will connect the matched processes using its link configuration table and inform the polling process. Then, the polling process can inform the connected parties about the readiness of the connection when the next polling turn comes for the processes. In this way, the *System Loop* and domain loops will not be used for the exchange of datagram packets to establish a connection, and the cost will be reduced considerably. The *System Loop* and domain loops will be used for only real data exchange. The IPC predictability also will be improved as the polling period of all nodes can be estimated in advance precisely.

The only problem with such a schema is that the number of Transputer links are not enough for setting point-to-point connections between processes after dedicating one of the links for connection establishment.

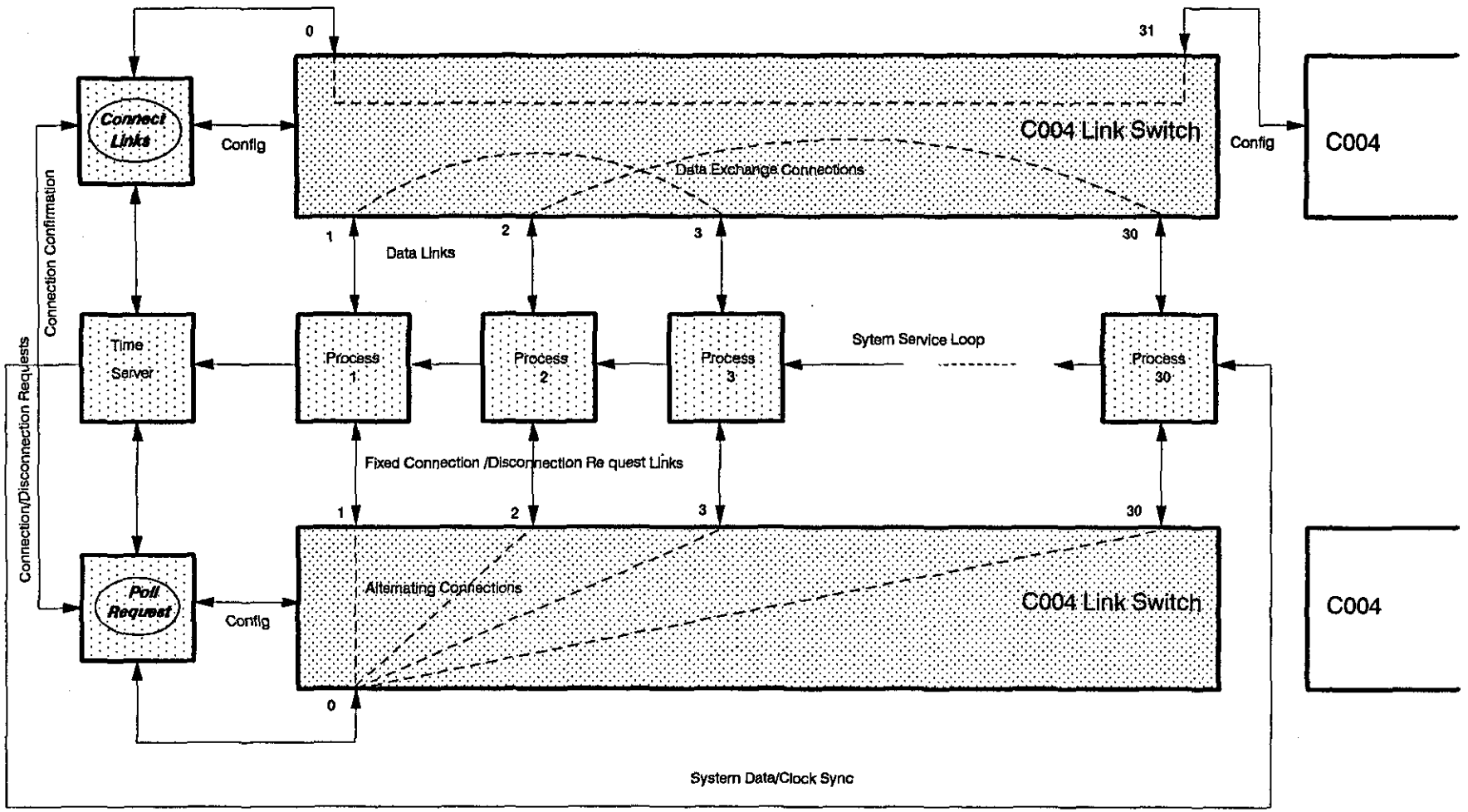


Figure 6(1) : Request Polling Technique for Connection Establishment

The new high performance (Transputer compatible) *Texas Instruments' TMS320C40* chips [TranstechA 1993, TranstechB 1993], with 6 serial Inmos compatible links, could be very suitable for such a schema. With the current Transputers, for this implementation domain manager nodes would need to be created using at least two Transputers, and this will complicate the design considerably.

The second optimization to reduce the communication cost of the RT-DOS processes is to group system services further into subdomains, similar to the current application domains. As the RT-DOS kernel and the IPC treats all system processes (including servers) and user processes in the same way, the grouping of the related system services in separate domains will isolate their message traffic from irrelevant messages, and as a consequence, lead to a better utilization of the system resources.

For example, grouping all terminal servers in a separate domain will contain multicast messages sent for terminals to this domain only. To handle multiple copy update problem also this approach will be very effective, as a multicast message can be sent to all copies of file servers in the same domain without affecting the network traffic on the other parts of the system. Please see **Figure 6(2)** for the suggested schema.

The idea of exchanging data between processes by only preestablished point-to-point connections is based on the following assumptions:

a) Processes don't exchange data arbitrarily. Instead, once a connection between two process is established they will keep exchanging data at least for a reasonable time period. Both of them will not need any other connection with a third party, as otherwise a separate connection setting per message exchange between processes would be necessary;

b) Support for broadcast and multicast class of message exchange will not be crucial at all. Either they will not be needed at all, or only very occasionally.

It can be shown that these assumptions are not valid by a few scenarios as below:

1) The status of any application domain (or process) might be monitored from a few terminals at the same time, or the status of a few application processes might be monitored on the same terminal alternately; both cases need to set a new connection for every data exchange, because the number of Transputer links to be used for connections is limited (maximum 2);

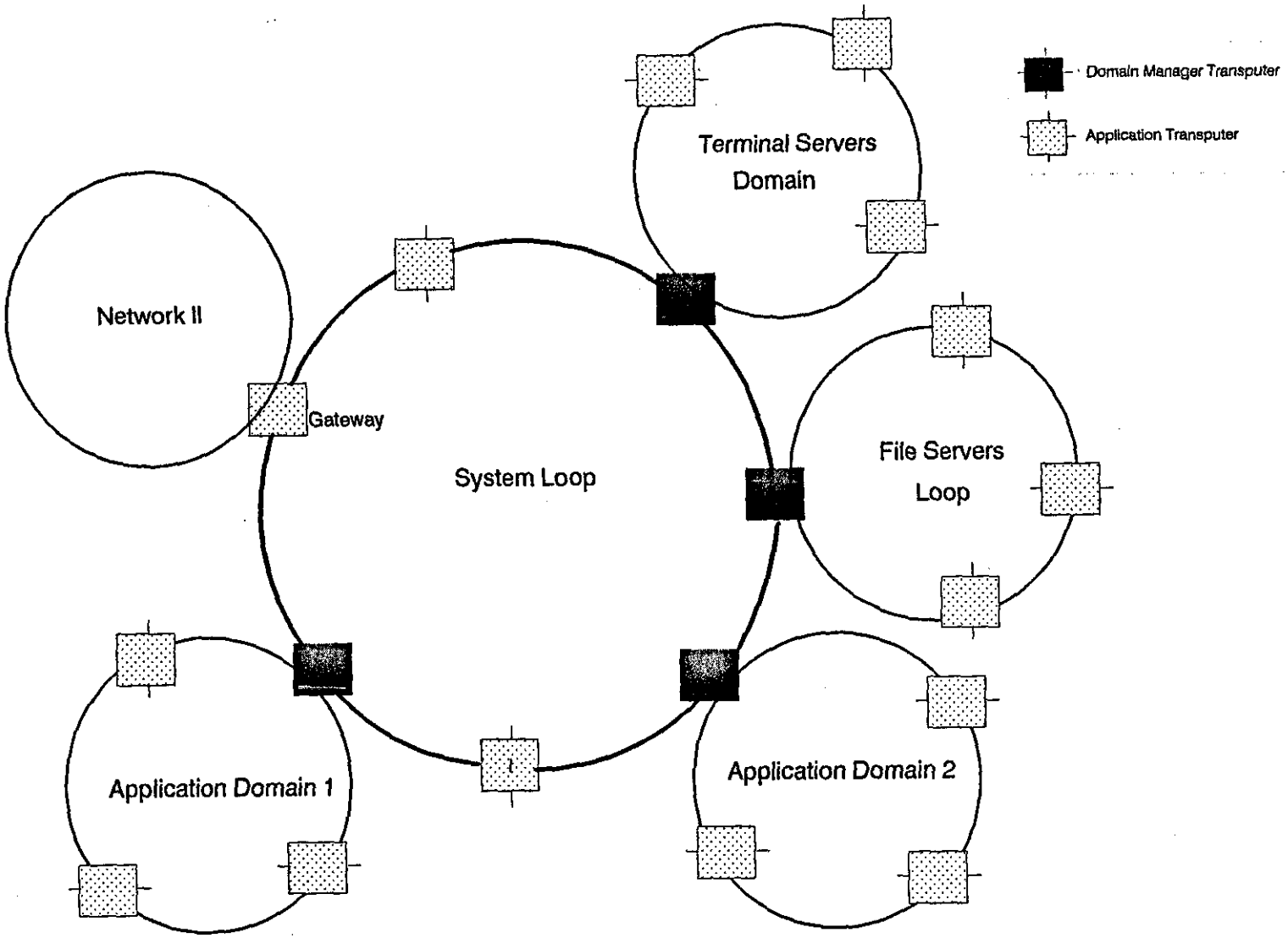


Figure 6(2) : System Services Further Grouped into Domains

2) In the same application domain, a process is receiving data from two remote (in the same domain) processes and sends data to another two processes, so that it needs to share its two data links for four different concurrent connections. This situation will impose the setting of a new connection for every data exchange. It means that, a process can't exchange data efficiently with more than one process concurrently;

3) A server process might be the destination of more than one process at the same time (for example a file server is supposed to handle a number of connections concurrently). No process can keep its connection alive for even a short period of time, because the server has to handle alternating requests of processes. Every data exchange with such a server process will cost a new connection setting for the server;

4) Keeping relations between processes stable and fixed (to increase performance) will decrease flexibility and concurrency of the overall system;

5) If any process wants to broadcast a message on all terminals, it will be too costly, if possible;

6) The aim of gathering interrelated processes in to the same domain is to isolate the domains from rest of the system to obtain error containment and increase the performance of the individual domains as well as the overall system performance (by reducing the intra-domain and inter-domain message traffics). This means that, processes in the same domain are likely to exchange data between themselves rather than between inter-domain processes. A datagram-like connectionless message exchange protocol might be more efficient than setting connections between processes (by accessing a remote switchboard process) for every message exchange. If some processes in the same domain need permanent (or near permanent) connections, permanent links can be established;

7) Datagram-like message exchange mechanisms can not be protected from user errors (users might increase message traffic accidentally or intentionally). On the other hand, the same condition is valid for point-to-point type of message exchange protocols. If two different processes set and reset connections with a third process concurrently in a loop, even without any real data transfer, the whole network can be brought to a halt by only datagram messages sent to establish these alternate connections.

The switchboard and connection service can be a bottleneck for point-to-point connection based protocols, as all connection requests should be handled by a centralized mechanism. All local processes in a domain must also send connection requests to the connection service, increasing the message traffic of the overall system unnecessarily. Multicasting and broadcasting will be needed in urgent situations to distribute a message at the same time to all parties. Point-to-point connection oriented protocols will decrease the efficiency (and effectiveness) of group communication services and reduce the overall system parallelism.

Nevertheless, the message switching protocol might not provide a predictable message delivery service (in the situations where group communication is a required) which is required by real-time systems. As a result, the following schema is suggested (and also adapted in the current IPC implementation):

a) Message exchange between processes by setting point-to-point connections should be done for bulk data transfer. Short messages should be exchanged by *datagram-like connectionless transport services*. When any processor pair needs continuous message paths a permanent connection between them should be established. In addition to this, conversation type of connections can be implemented as an upper level layer on top of the datagram services to be used optionally when it is needed, instead of forcing all users to have only point-to-point connection transport protocol. Some processes might not need connectionless protocols at all;

b) Predictability and reliability issues should be handled in upper layers to keep the IPC size at a minimum;

c) Responsibility of separating bulk data transfer from small messages will be the responsibility of upper services in collaboration with kernel;

d) Reliability issues should be handled by the policy layer end-to-end protocols established between the user processes themselves during the setting of the sessions.

Some improvements can also be made to the current implementation. One of them is using a schema to redirect the connection request of the initiator process to the connection server directly by the designator, as is shown in **Figure 6(3)**.

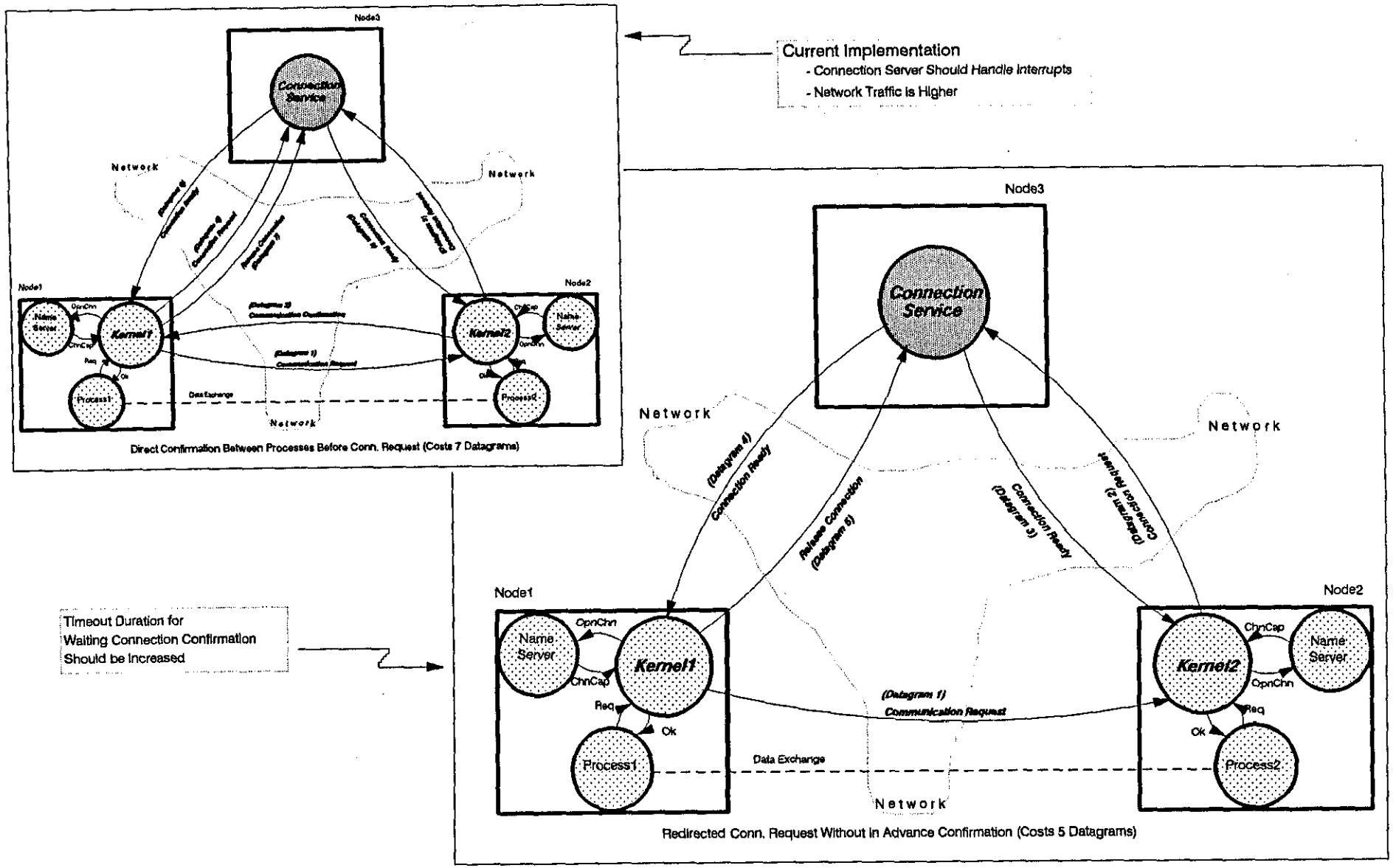


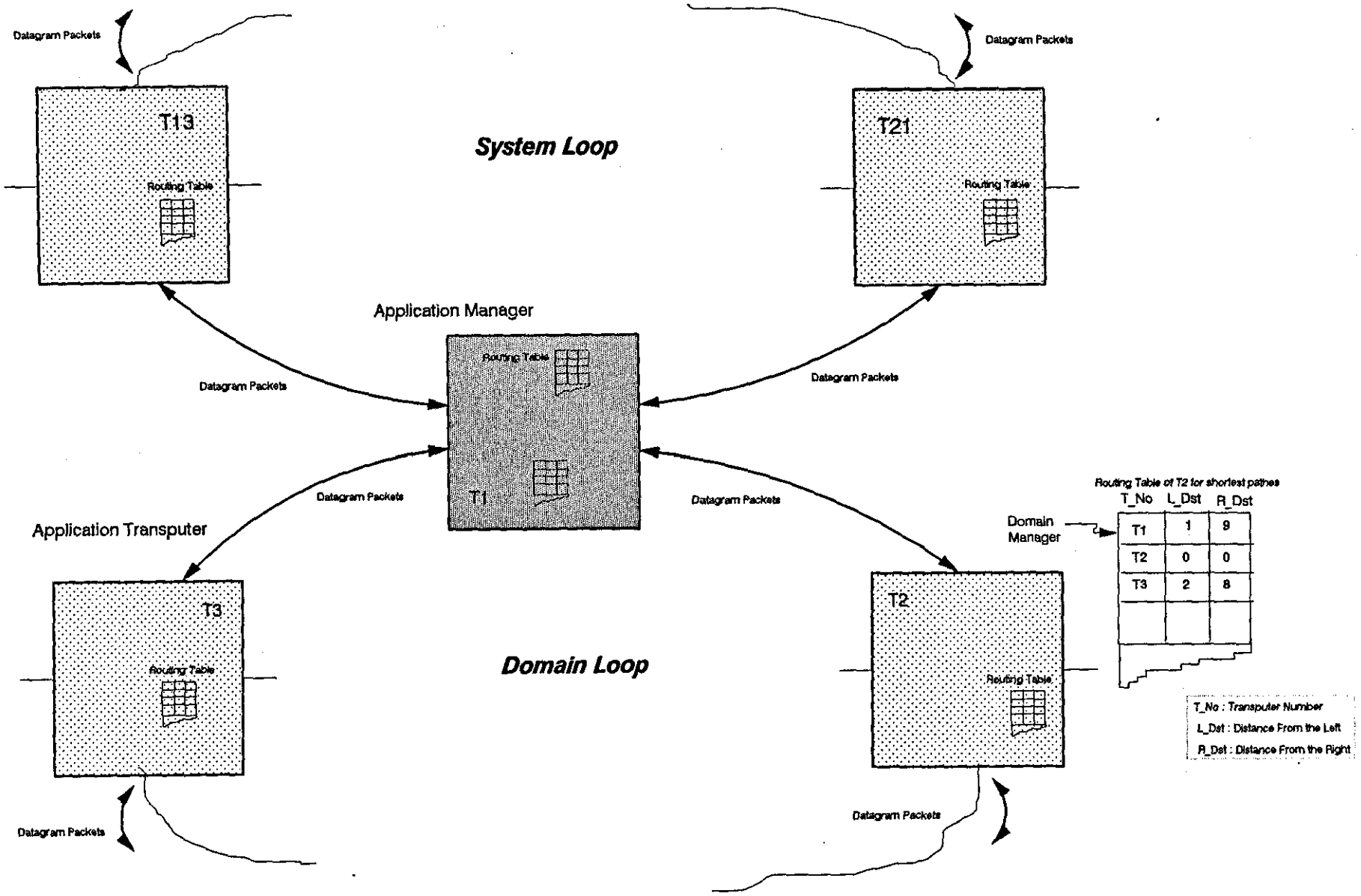
Figure 6(3) : Redirection of Conn.Requests to Connection Service by Destinator

In this schema, if the destination process is ready for forming a connection, instead of sending a reply to the initiator process about acceptance of the communication it matches the request and forwards it to the connection service. This will reduce the number of datagram packets to be exchanged from seven to five only.

The rest of the connection procedure would be the same as current implementation steps: connecting Transputer links, informing communicating parties, and releasing the connection after data exchange.

However, the most important disadvantage (and the only negative affect on the current implementation) related to this approach is the necessity of relatively longer time-out periods. As the IPC relies on time-out mechanism for resolving deadlock conditions, it should wait longer than the current implementation just to be sure that connection request of initiator is not lost; because, the reply of the responder will not directly come to the initiator. Instead, it will go to the connection service agents first, and after connection is set, the switch manager will send a notification to the initiator to inform about readiness of the physical links. If a connection request is lost, or some how did not get confirmed immediately, the initiator should wait for enough time to be sure that connection request is not under process by one of the connection service managers.

The second improvement which can be added to the current implementation is that routing algorithms can be made more efficient by modifying as shown in **Figure 6(4)**. A short-cut (or shortest path) algorithm can forward packets from both directions of the loops (*Application* and/or the *System Loop*) instead of a fixed direction, considering the number of the nodes to be hopped in each direction. To do so, each kernel should maintain a shortest path table on each node to indicate the path length of each destination from the current node in both directions. At any node, the routing algorithm will check the routing table before forwarding a datagram packet to choose the direction through which the packet can reach its destination earlier, and send the packet through the relevant link of the chosen direction. Application manager nodes should maintain two of these tables, one for the related domain loop itself and one for the *System Loop* on which the domain manager node is residing. If a packet should be sent through the domain loop the table to be used for the shortest path direction is the one which maintains distances on the domain loop.



Figue 6(4) : Routing Tables of Short-cut Algorithm for Datagram Packets

The most important problem related to this approach is that maintenance of these shortest path tables could be very costly, especially if application domains are built (dynamically) very frequently (this cost can be ignored if the system topology is reasonable static as the tables are built once during initialization only). In addition to this, for most of the group communication protocols (broadcasting an multicasting) this algorithm will be useless as the packet should visit all nodes on the network regardless of their physical location. In static topologies, as the tables do not need to be updated very often, they can be built during the system initialization (initial loading of configuration tables) and successfully used especially for the unicast packets.

In the following section, these issues are elaborated in more detail, as some of the options has been implemented for testing their performance affects on the IPC mechanism.

7 EXPERIMENTAL RESULTS AND CONCLUDING REMARKS

In this chapter, the experimental tests which have been conducted to evaluate the real performance and behavior of the IPC under different circumstances, are discussed presenting the results with graphic charts. The possible reasons for the obtained results are explained and the relevance of the different communication parameters in predicting certain system behaviors are elaborated. Finally, the real lessons that have been learned from the experiments are briefly submitted at the end of the chapter.

The hardware configuration of the testbed which has been used for implementation of the RT-DOS IPC mechanism is shown in **Figure 7(1)**. As it can be seen on the figure, 21 Transputers of different types (**T425**, **T414**, and **T212**) are connected each other to form a system loop and two application domain loops. The development system Transputer (Host Transputer) is excluded from the loops and used only system development and network loading. This node is being used as a system monitor as well, during the tests.

7.1 Experimental Results, Performance and Behavioral Considerations

The software configuration of the testbed (the hardware topology has been given in **Figure 7(1)**) which has been used for measurements to evaluate the IPC performance and identify the system behavior under different communication loads is shown in **Figure 7.1(1)**.The testbed consists of two application domains of 10 and 2 Transputers each, and a system domain of 10 Transputers. One Transputer (T414) is reserved for host operations, such as initial file loading, user/system interaction, tracing, etc.; and a process (**HostP**) is placed on it to perform the aforementioned functions. A T212 Transputer (16 bit) is dedicated for dynamic link switch operations and a process (**Switcher**) is placed on it to carry out the physical link connection establishment task. The rest of the physical testbed can be summarized as follows:

- Speed of all links between Transputers of application and system domains were set to 20 Mbits/sec;
- Switched links were controlled by a single (centralized) Connection Server process (**ConnSrw**) which is located on one of the system domain nodes, and consisting of a single link switch hardware controlled by a T212 Transputer. This node is used as the initial network loading point as well, as it is connected to the host Transputer physically through one of its links;

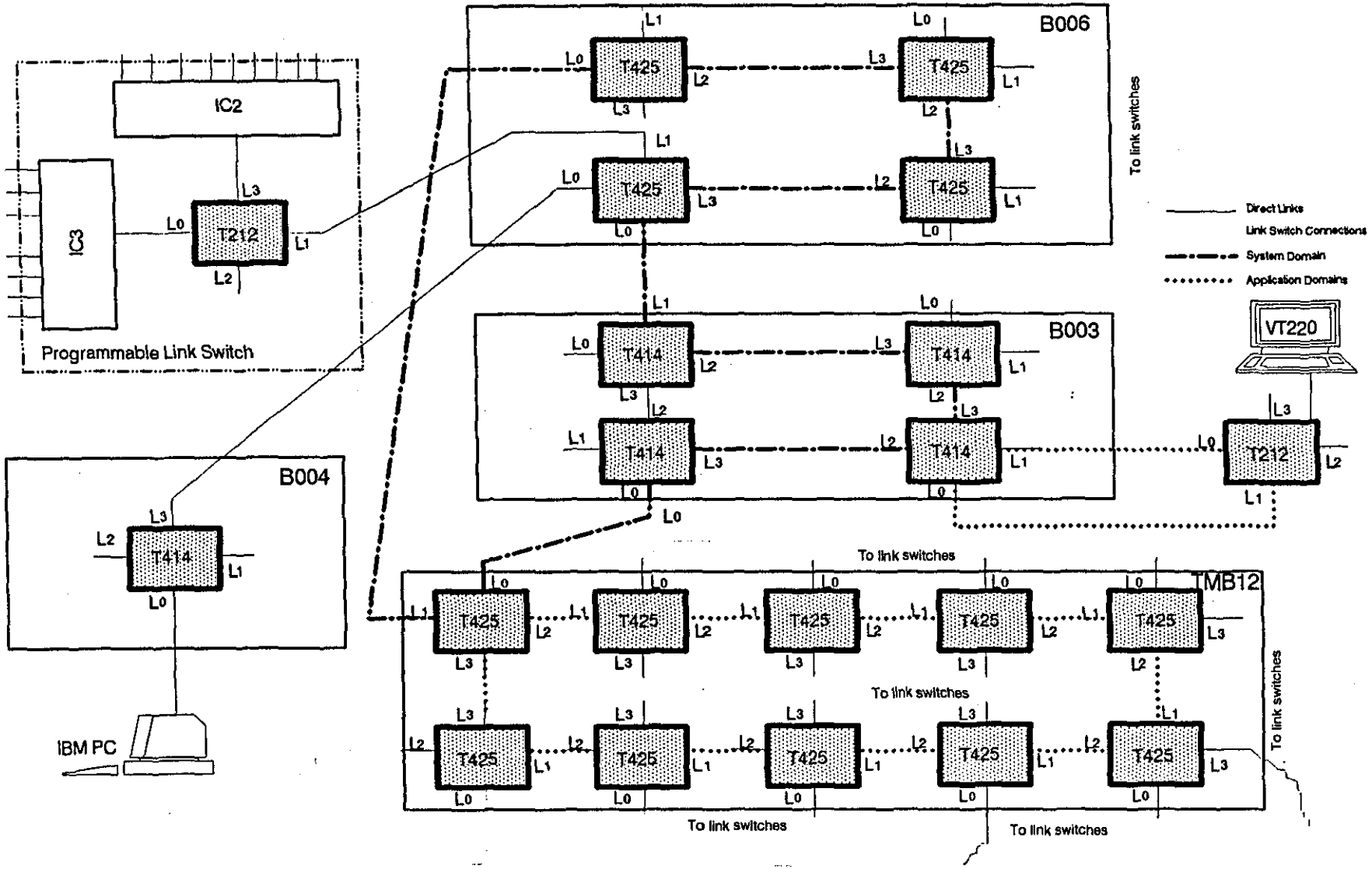


Figure 7(1) : Hardware Topology of the RT-DOS Testbed Used to Measure the IPC Performance during the Implementation

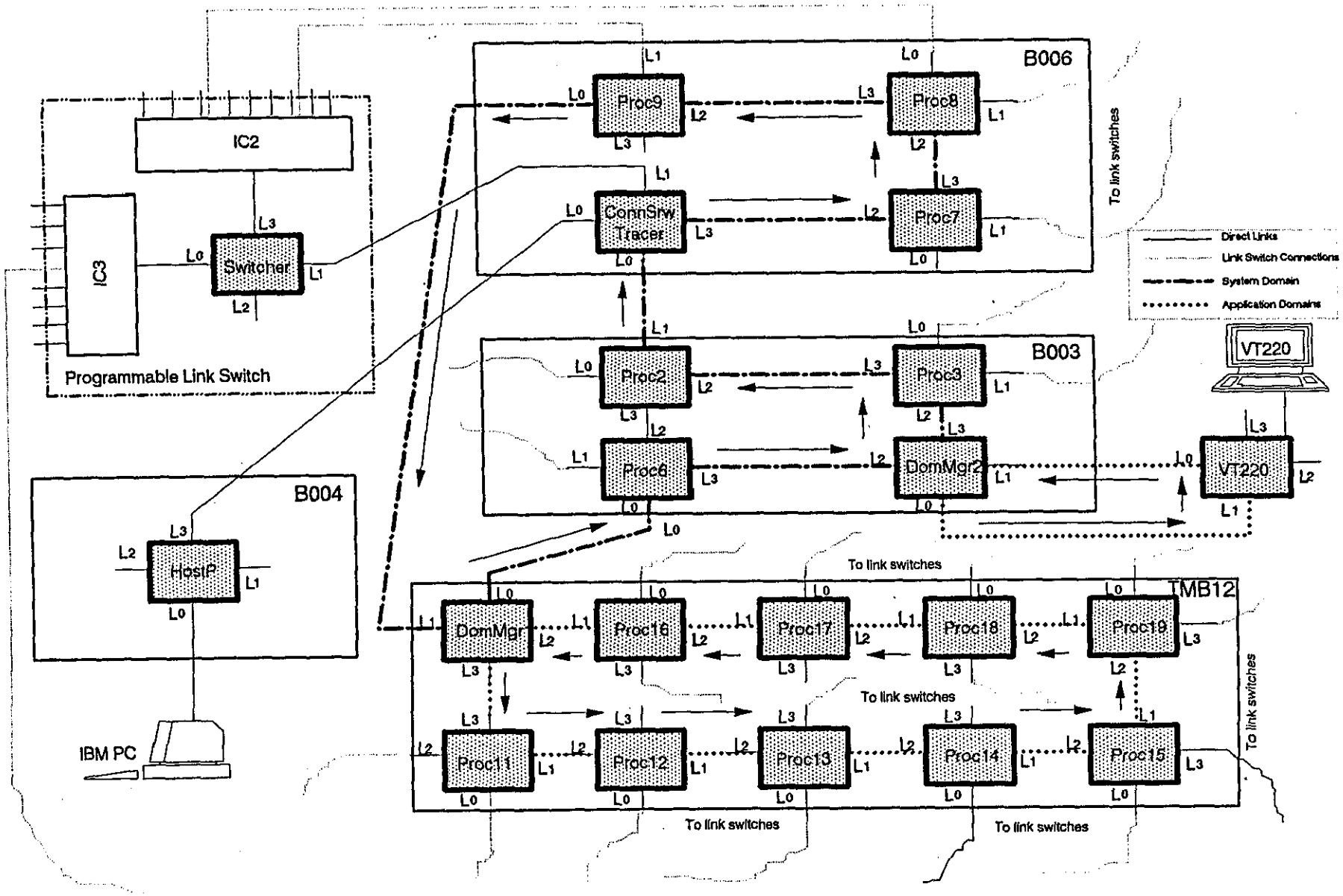


Figure 7.1 (1): Physical network node locations of processes which generate varying IPC communication load conditions under which measurements have been conducted.

- A tracer process (**Tracer**) was also placed on the same node at which Connection Server process resides, to display the experiment results on the host terminal screen;
- Processes (**Proc2-Proc19**) were placed on all the remaining Transputers to generate varying conditions of message traffic; two domain managers were not involved in any communication interaction except for routing/forwarding the coming/going messages;
- All Transputers involved in communication are either 15 MHz T414 or 25 MHz T425 Transputers, except the one which is connected to a VT220 terminal (it is a T212);
- Tests were conducted with the version of IPC which was designed and implemented with the group communication protocols in mind. The shortest-path algorithm was not in effect during tests as it wouldn't make any sense to use such algorithms in group communications since the destination process addresses are not exactly known in advance.

The objectives of the experimental tests were to investigate:

- Connection times (minimum, maximum) between two processes placed on varying physical network locations (distance from Connection Server process) under different communication loads; and predictability limits of the IPC in terms of connection establishment times, under certain conditions;
- Minimum and maximum datagram delivery times (and node traversal times) of a datagram under different communication loads; and predictability limits for connection establishment and message delivery times;
- Effect of message size and/or IPC datagram buffer size in IPC performance and predictability limits;
- Performance changes in group communication protocols in terms of message delivery times, under different communication loads with varying number of destination processes;
- Communication overhead of group communication protocols and their effect on IPC predictability limits.

Considering the above objectives, a number of tests are conducted. The best and/or the worst communication conditions, which are most ideal for each communication type for

which the test have been performed, were generated with different combination of processes. The facts that have been observed are presented in this chapter by a number of graphic charts, and each result is interpreted briefly to elaborate on the possible reasons.

The most interesting result of the tests was that communication delay of a message did not increase significantly with the increased number of destination processes in multicasting, as can be seen in **Figure 7.1(2)**. During this test, a message with a size of 2000 Bytes was broadcasted from **Proc11** (see **Figure 7.1(1)** please). The internal Transputer times elapsed for the message to reach a number of different destination processes and get the reply (2000 Bytes) back to the same source node from all of these destination processes were also measured. Communication delay of a message broadcasted to 14 destinations (28200 μsec) was less than one third of one single message circulation time (21300 μsec) spent for the same distances.

Physical locations of destination processes did not make any important performance differences during the test, as all messages should almost travel through the same number of network nodes to reach to the message source back. The other reason for this result was that the processing power of the nodes which are closer to the **Tracer** process (15 MHz T414 Transputers) were less than the power of the remote ones (25 MHz T425 Transputers). Under the physical capacity limitations of the current system, this behavior of the IPC multicasting service was consistent.

Obviously, this behavior supports a predictable message delivery service for group communication protocols. This can provide, in turn, a *predictable multiple copy update facility* for implementation of fault tolerance systems in distributed environments, with an acceptable communication overhead and minimum performance degradation. With point-to-point communication protocol (circuit switching) the cost would multiply almost linearly with the increasing number of message destinations, as a separate connection should be established with each destination to implement the group communication service.

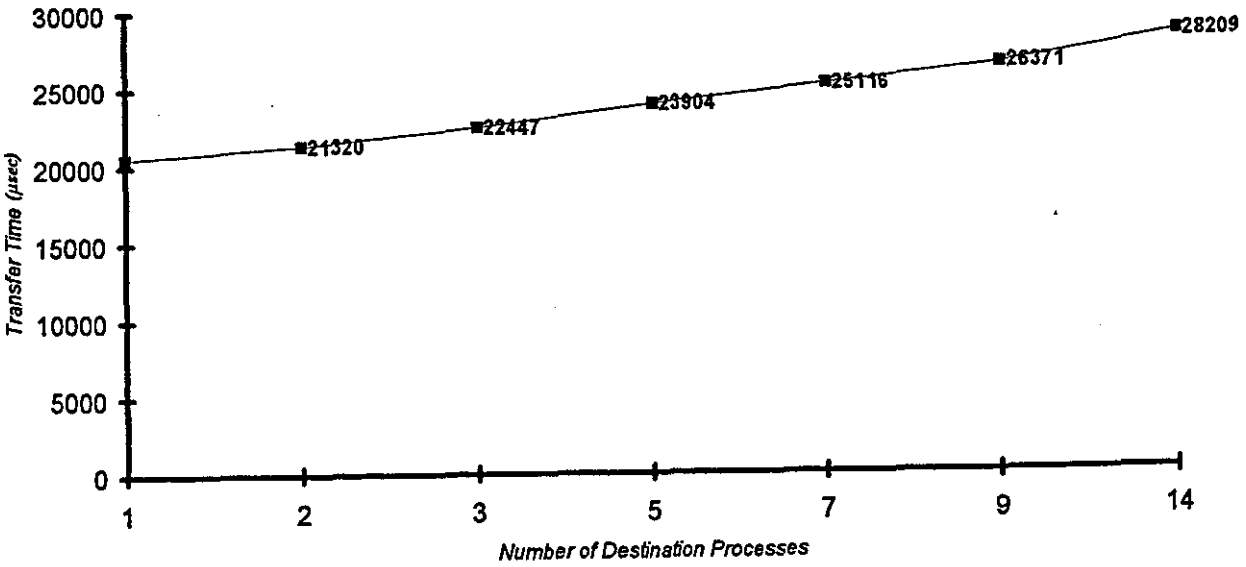


Figure 7.1 (2) : Average multicasting time (μsec) of one message from the same source node to a varying number of destinations.

To investigate the effect of IPC datagram buffer size and message length on the IPC performance and physical capacity limits, a number of tests were conducted. The results are shown in **Figure 7.1(3)**, **Figure 7.1(4)**, and **Figure 7.1(5)**.

As it can be seen in **Figure 7.1(3)**, the average transmit time of one datagram from **Proc11** to **Tracer** process did not change considerably for 5 to 40 buffer limitations. **Proc11** transmitted 2000 byte messages to **Tracer** process (14 nodes apart) continuously, and the messages were displayed on the host terminal by **HostP.** Because of the memory restrictions of the testbed Transputers, buffer size couldn't be increased beyond 40. The worst performance was obtained with 2 buffers (almost 30% reduction comparing 5 and more buffer limits), while 20 buffers gave the best performance.

This is, most probably, because there are 5 concurrent processes running independently in the IPC mechanism to handle the datagram routing at different points of the nodes. To reduce the buffer size to less than 5 makes these processes wait for each other and reduce the concurrency of the parallel IPC modules. The best performance was obtained with 20 datagram buffers. After that point, the manipulation of longer buffers starts consuming relatively more node processing time, and reduce the amount of real communication. The number of datagrams which could reach to the **Tracer** process (which in turn displays them on the host screen) in one second was (maximum) 290 with 20 buffers under the above conditions, as can be seen in **Figure 7.1(4)**.

Another test was conducted to investigate the effect of message length on performance of the IPC datagram services. One single message of varying length was sent from **Proc11** to **Proc9** and received back by **Proc11** again (each message visits 19 nodes). The results of the test can be seen in **Figure 7.1(5)**. The total communication cost of a message with a length of 2000 bytes was only 3% more than the communication cost of a message of one byte length, for the same distances.

A similar test was conducted by using the nearest nodes to the **Tracer** process as the source of datagrams, with a message length of 40, 100, 200, 500, 1000 and 2000 bytes. The average message transmit times for these message lengths can be seen in the **Figure 7.1(6)**.

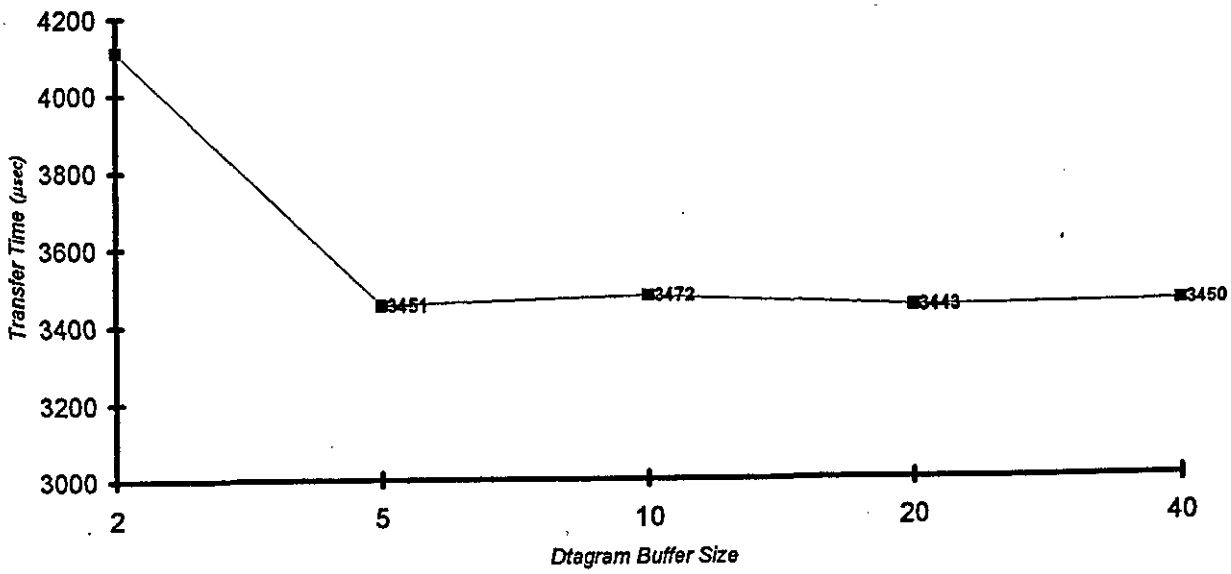


Figure 7.1 (3) : Average traversal time (μsec) of one datagram (2000 Bytes) from the same source to a fixed destination under varying IPC buffer size limitations.

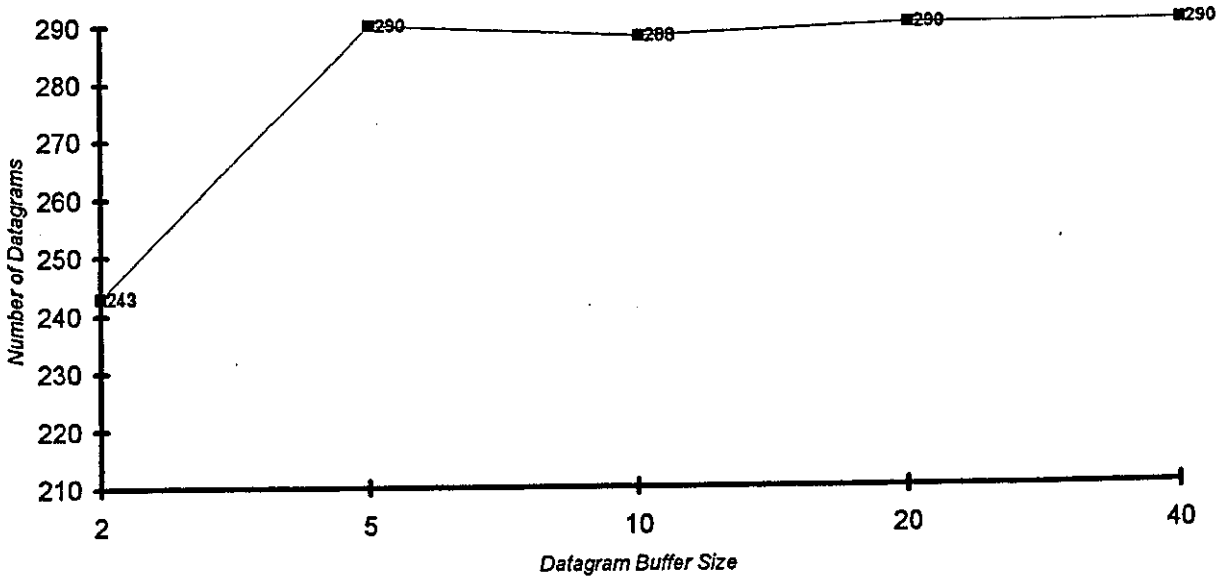


Figure 7.1 (4) : Maximum number of datagrams traveled through 14 nodes (from Proc11 to Tracer) under different IPC datagram buffer limitations (message size = 2000 Bytes).

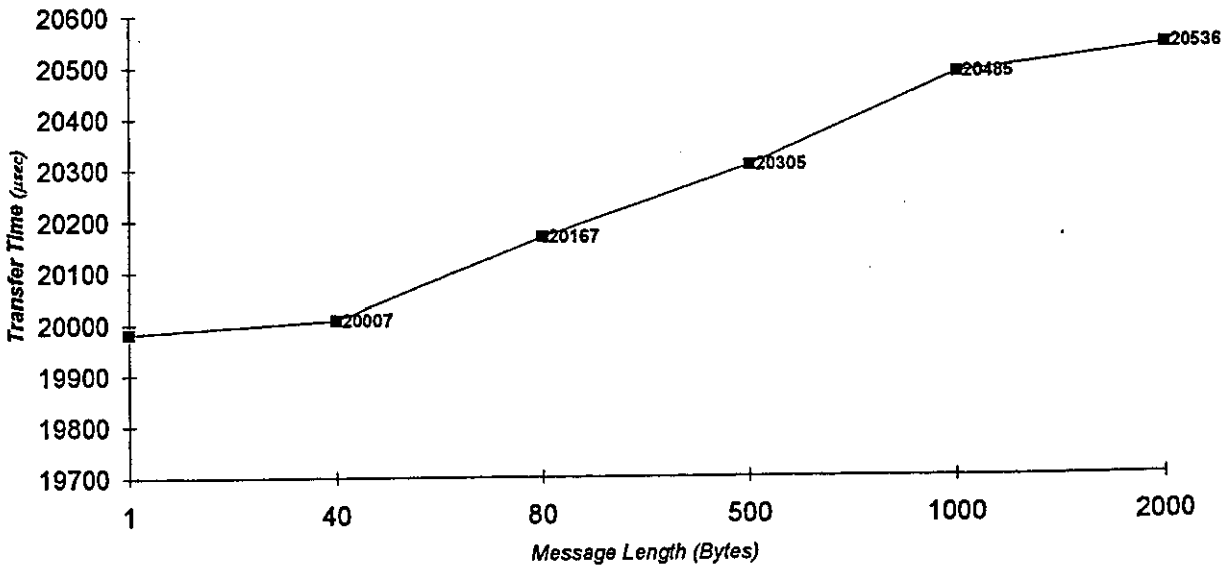


Figure 7.1 (5) : Average traversal time (μsec) of one single message in varying length through 19 nodes.

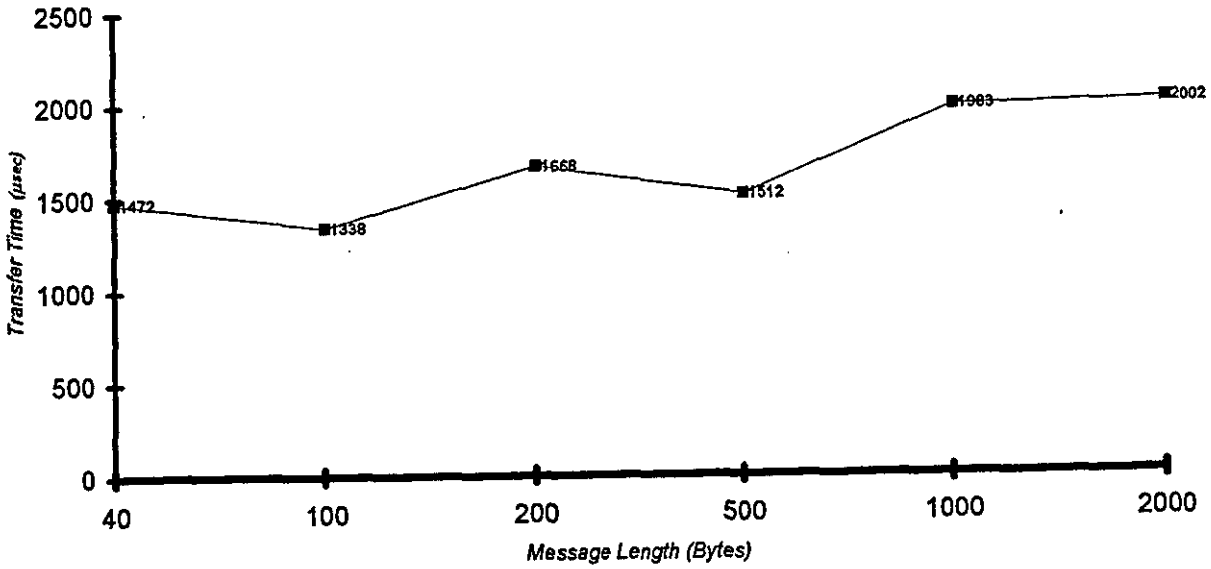


Figure 7.1 (6) : Average traversal time (µsec) of a datagram with a message length of 40, 100, 200, 500, 1000 and 2000 bytes, through 3 nodes.

Here, the reason for the difference between the transmit times of 40 and 2000 bytes of messages being relatively bigger than the previous test (as shown in **Figure 7.1(5)**) is that the effect of total node processing overhead of each message is less than the previous test, because each message is manipulated three times in the previous test while it is processed only once in this test. Hence, the net effect of the message transmission times through links is becoming a dominant factor with more effect on the total IPC communication.

This results can be attributed to the fact that the efficiency of the IPC protocol implemented for a network of Transputers is not determined by the transmission time of packets, but rather by the processing time spent at the nodes. This claim has already been proven in a similar research study [Waring 1990]. In this research, the times taken for the return journey from the manager to a remote processor were measured for message lengths of 1, 16, 128 and 1024 words. For their IPC shell, these times were 1, 2, 5 and 30 units respectively; implying that communication overhead increase is at least relatively 10 fold less than the increase in datagram message length for Transputer link communication.

The reason of the more efficient results we got for the longer messages is that the time spent on node processing is considerably higher than the link transfer times in our case. This is because of higher node interrupt overhead for the multi-functional structure of the RT-DOS IPC, reducing the effect of the link communication delay on the total communication overhead. The reduction in computational capacity of the nodes does not reach to more than 10% of the total node computational power for unbroken streams of packets which are bigger than 128 bytes. For smaller packets, the initial fixed cost of processing and switching times is becoming the same (or more) as the time taken to transmit a single packet.

That means that, the minimum packet length should be more than 128 bytes to be able to utilize the full communication capacity of Transputer links. The effect of through stream traffic becomes important only when the packet length is very short for a network of Transputers.

Another similar test, using only two different length of messages (40 bytes and 2000 bytes), was also performed to investigate the performance effect of the physical network locations of processes, on message delivery.

As it can be seen in **Figure 7.1(7)**, four processes (**Proc11**, **Proc19**, **Proc6** and **Proc2**) of varying distances from the same destination (**Tracer**) were continuously sending messages to the **Tracer** process to generate through network traffic, under the limitation of 40 IPC datagram buffers. Average message transmit times (in microseconds) for these sender processes were measured. The performance difference for the two message sizes was consistently very close for almost all source nodes, except the nearest source (**Proc2**) being relatively more advantageous than remote nodes for 2000 byte length of messages.

These observations can be interpreted as follows:

- For less than 2000 bytes of message length, performance effect of message length on IPC message delivery service can be ignored; and below the physical system capacity, predictability of message passing service is consistent;
- If the first assumption is true, then a nearly six times more efficient communication service (without sacrificing the predictability of IPC) can be obtained when message switching is used instead of point-to-point connection protocol for a data exchange of less than 2000 bytes, as a minimum of six datagrams should be introduced to the system to exchange only a message of the same size with the point-to-point connection establishment service;
- To utilize the full communication capacity of Transputer links, a message size of at least more than 100 bytes should be used.
- The above conditions imply that, for the environments in which 90% of the messages exchanged between processes are less than 1KB (which is very likely for the majority of real-time systems), the message passing is a more efficient (cost-effective) communication mechanism which provides predictability at higher precision, and yet, supports group communication protocols (which, in turn, support the implementation of fault tolerant systems) with a minimum overhead.

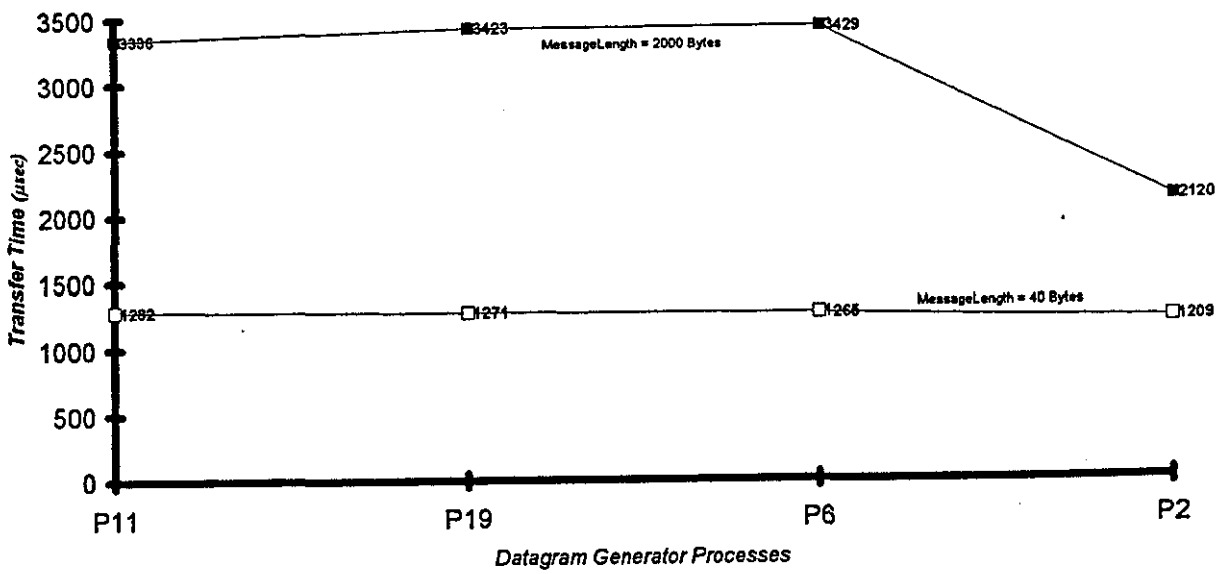


Figure 7.1 (7) : Average traversal time (μsec) of one message in varying length from a number of source nodes with differing distances to a fixed destination node.

Maximum and minimum connection establishment times and predictability limits of the system under varying communication loads were also measured. The test was performed with a number of combinations of processes at varying distances from each other, with a limitation of 40 IPC datagrams per buffer under different connection request loads. System capacity was reached at about 400 connection requests per second, and providing that this physical capacity limitation is preserved, a predictable connection establishment time was obtained regardless of network locations of process pairs.

The minimum and maximum connection establishment times (10309 μsec and 16129 μ sec) can be seen in **Figure 7.1(8).** Maximum and minimum number of connections which are established under different request loads were 97 connections/sec and 62 connections/sec, respectively; as it can be seen in **Figure 7.1(9)**. Beyond the physical system capacity of 400 connection requests per second, predictability of the connection establishment time is not guaranteed, as it increases exponentially; and the number of connections which can be established in one second is also decreasing rapidly. Factors contributing to this results are obviously network capacity and connection server capacity. In addition, efficiency and performance of IPC modules and their implementation have a very important impact on the system capacity and predictability limitations.

Using the same testbed, hardware limitations and the topology, another version of the IPC was implemented with no group communications and/or message passing protocols in mind (only point-to-point communication protocol was implemented). To minimize the node processing overhead, most of the function calls were replaced by macro calls; to reduce the run time memory allocation/deallocation overhead, all datagram buffers and other system data structures were statically allocated during initialization time; and the Inmos semaphores were replaced by a more efficient version which has been implemented for the RT-DOS kernel.

A shortest-path routing algorithm was also implemented to minimize the datagram traversal times for connection establishment.

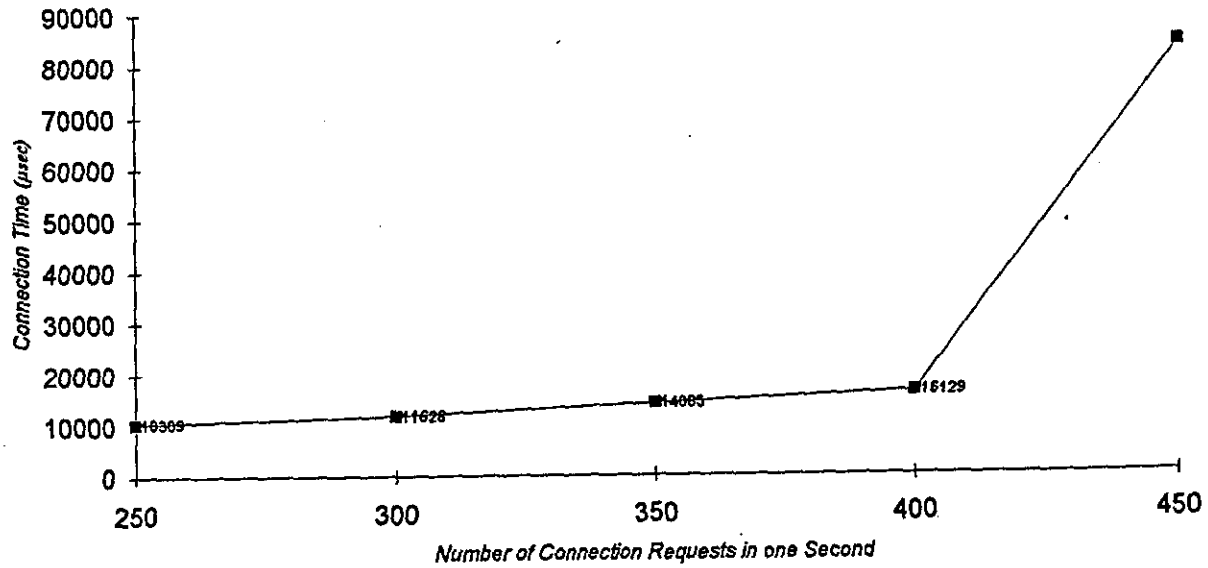


Figure 7.1 (8) : Average connection establishment times (μsec) of two randomly located processes under varying communication load conditions.

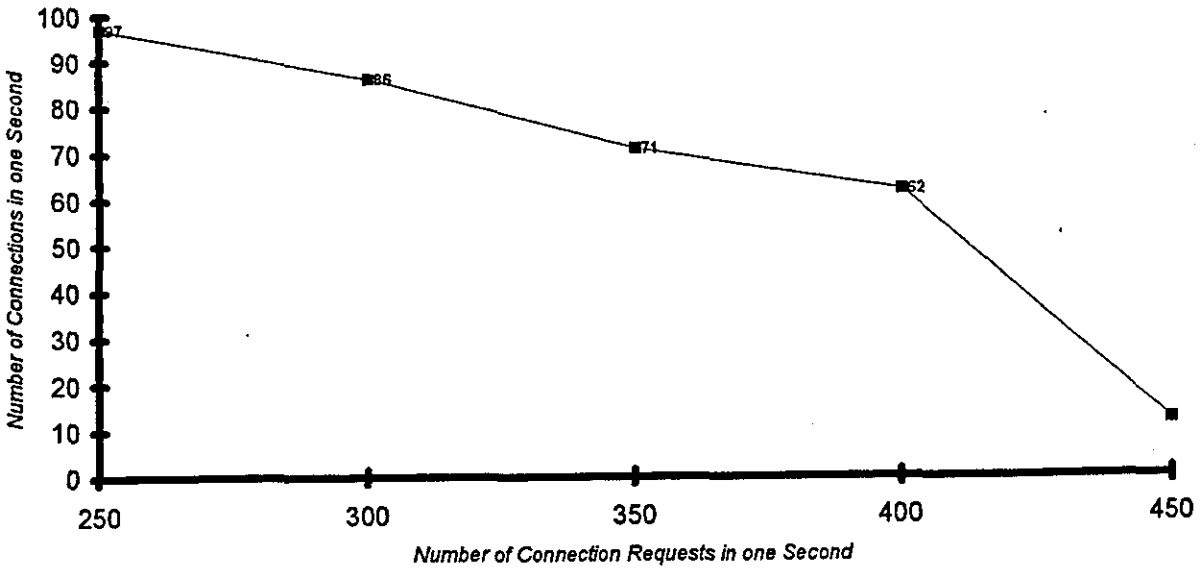


Figure 7.1 (9) : Average number of connections established by Connection Server under varying communication load conditions.

The results of measurements which were performed with this optimized version of IPC were published in a world Transputer conference [Benmaize 1993]. The net performance of the optimized IPC (in terms of the communication capacity of the system) is almost twice as much as the version introduced in here. The minimum and maximum connection establishment times were almost 4 times better, guaranteeing the IPC predictability. Predictability limit of the IPC was measured as 650 connection setup/second with a small variation in connection time. Minimum and maximum connection establishment times in predictability limits were 1.5 msec and 4.5 msec, respectively.

Another test was carried out to investigate the impact of different communication traffic loads on performance of IPC message delivery service, for a length of 40, 500 and 2000 bytes. During the test, the IPC buffer size was set to 40, and between 1 to 10 processes located on randomly chosen places, were sending datagrams at fixed intervals (from 10 to 64 msec) to the **Tracer** process.

The comparative performance results of the test for these three message length are shown in **Figure 7.1(10)**. The maximum message transmit times under the maximum communication load which is generated by a total of 9 nodes (11,000 datagrams per second) was 1852 μsec for a message of 40 bytes, and 2998 μsec for 2000 bytes. The minimum message transmit times for the same length of messages were 1472 μsec and 1983 μsec, respectively, under the minimum communication traffic of only one single producer process.

The node distances of the producer processes to the consumer process were found to be immaterial, most probably because of the difference between processing power of the T414 and T425 Transputers which are used in the testbed. As the less powerful T414 Transputers were more close to the **Tracer** process which has a physical connection with the host terminal, the message generation and processing times of the more distant processes (residing on more powerful T425 Transputers) were about the same as the closer ones. This observation also supports the idea of that the node processing time is a more important factor than the link communication time in determining the net IPC communication performance for network of Transputers.

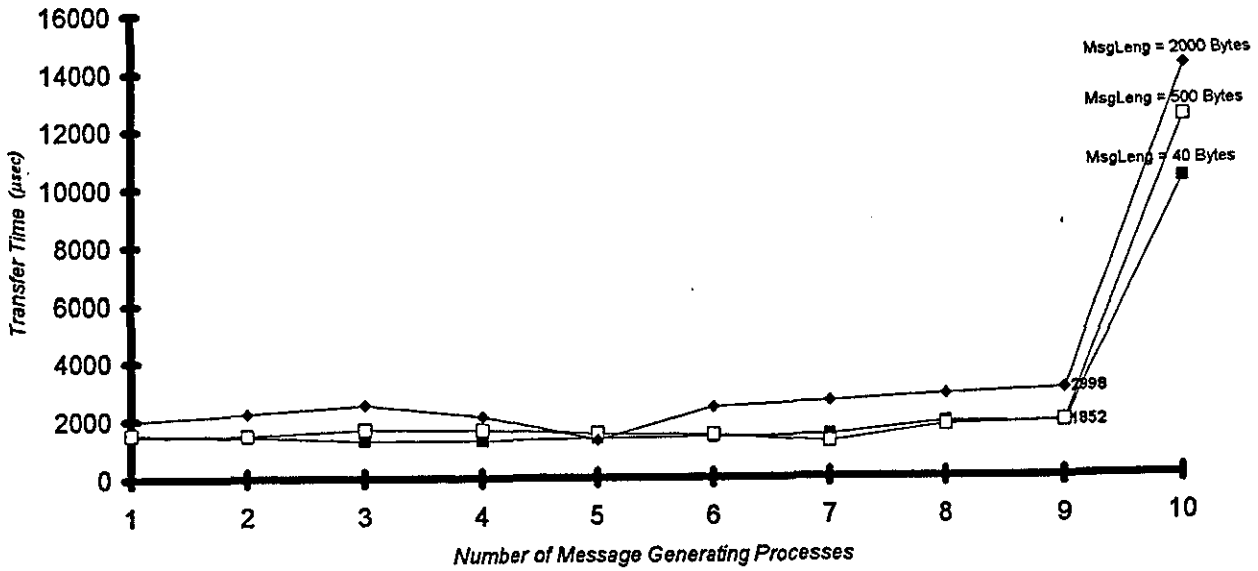


Figure 7.1 (10) : Average traversal time (μsec) of one message of varying length through 19 nodes under differing communication load conditions.

The minimum and maximum message transmit times under the message traffic of one single process (minimum communication load) to 9 processes (maximum communication load) are given in **Figure 7.1(11)**, **Figure 7.1(12)** and **Figure 7.1(13)**, for the message lengths of 40, 500 and 2000 bytes, respectively.

As can be seen from the figures, beyond the physical communication capacity of the system (the point at which 10th process also starts generating messages) average message transmit time is increasing very rapidly which leads to non-predictable system behavior in terms of connection establishment and message delivery times.

Another test was conducted to investigate the behavioral changes of the system beyond its physical capacity. **Figure 7.1(14)** shows the distribution of datagrams which were generated under maximum communication traffic load (beyond of the physical capacity limitation of the system) in a fixed time period, for the processes which reside on nodes with different distances from the destination process (**Tracer**). The test was performed with a message length of 2000 Bytes and 40 IPC datagram buffers. All processes were sending messages to the **Tracer** process continuously to be displayed on the host screen.

As it can be seen in the chart, **P6 (Proc6)**, **P3 (Proc3)** and **P2 (Proc2)** were more successful than the others in generating and sending a regular message traffic, **P2** being the most lucky one. This is because, these three processes are physically more close to the destination than the other ones, and the **P2** is the closest one. Beyond of the physical system capacity, because of the network congestion, message delivery or connection establishment times are becoming unpredictable, leading to an unpredictable system behavior.

The same test was performed under the traffic load of a combination of lesser processes (under the minimum communication load), with the other system parameters being the same (40 buffers, 2000 bytes of message length), except that processes were generating datagrams at certain internals (very small intervals such as 1 to 5 msec) to avoid over traffic and network congestion.

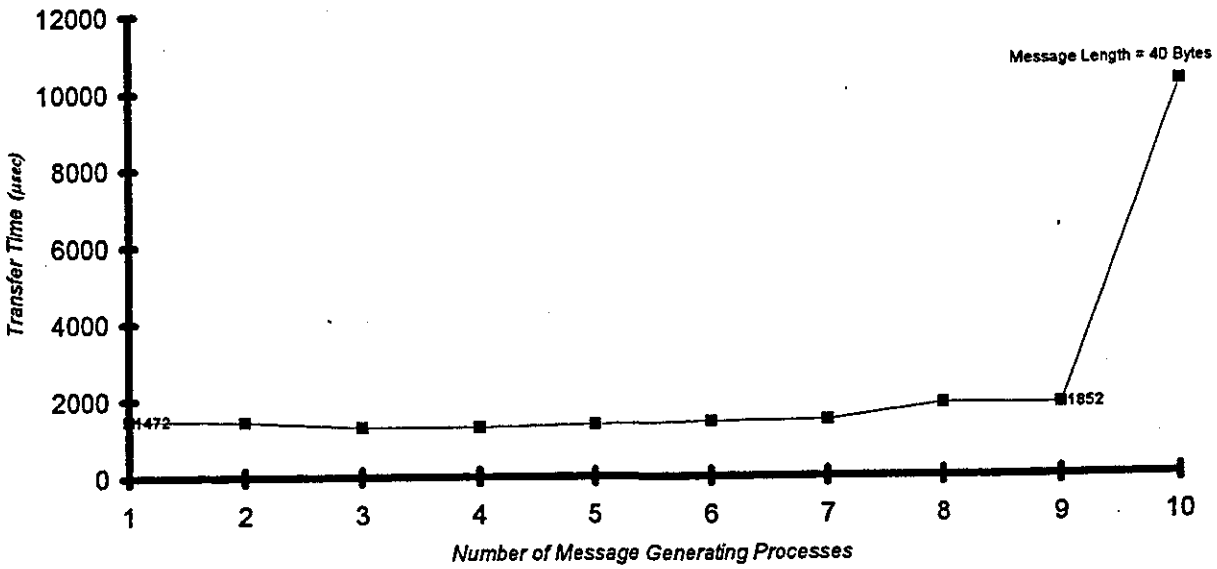


Figure 7.1 (11) : Average traversal time (μsec) of one message (of 40 Bytes) through 19 nodes under differing communication load conditions.

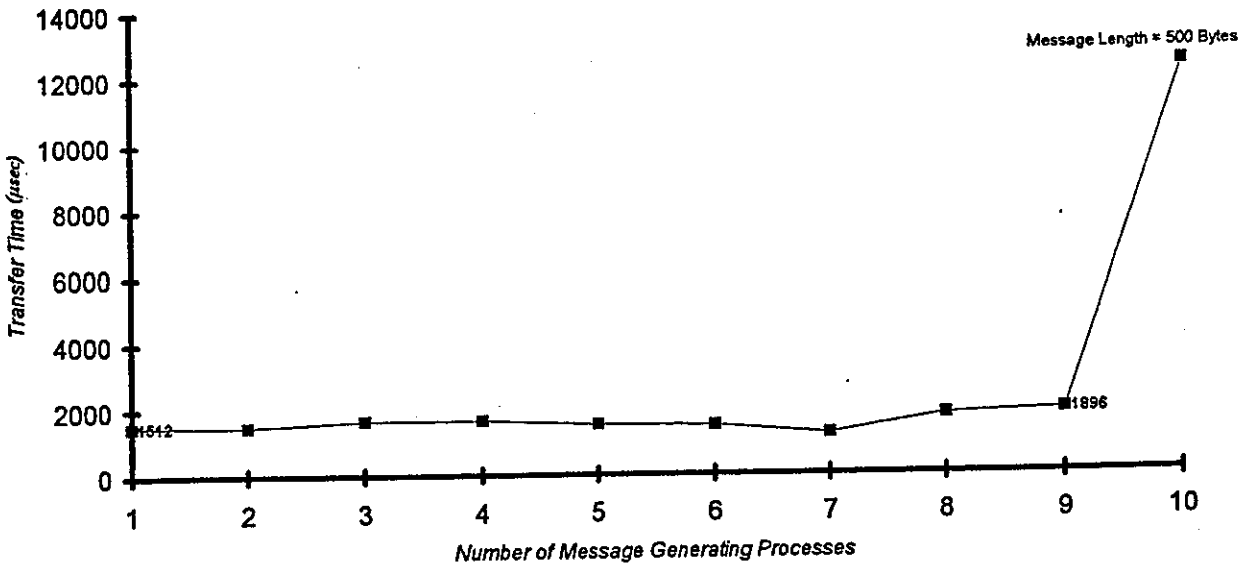


Figure 7.1 (12) : Average traversal time (μsec) of one message (of 500 Bytes) through 19 nodes under differing communication load conditions.

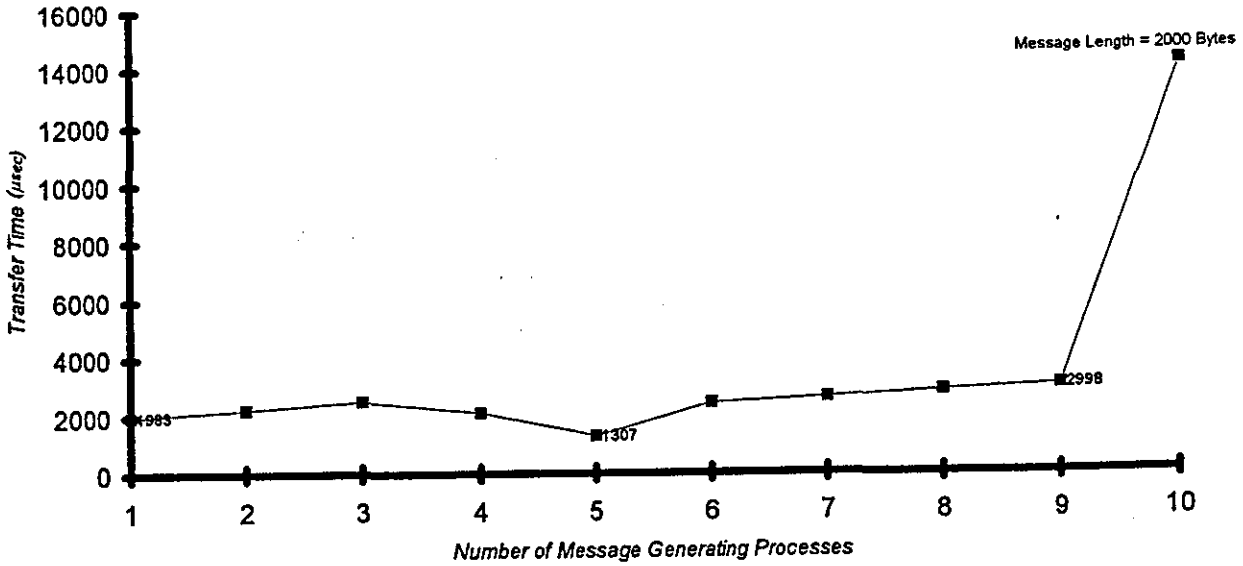


Figure 7.1 (13) : Average traversal time (μsec) of one message (of 2000 Bytes) through 19 nodes under differing communication load conditions.

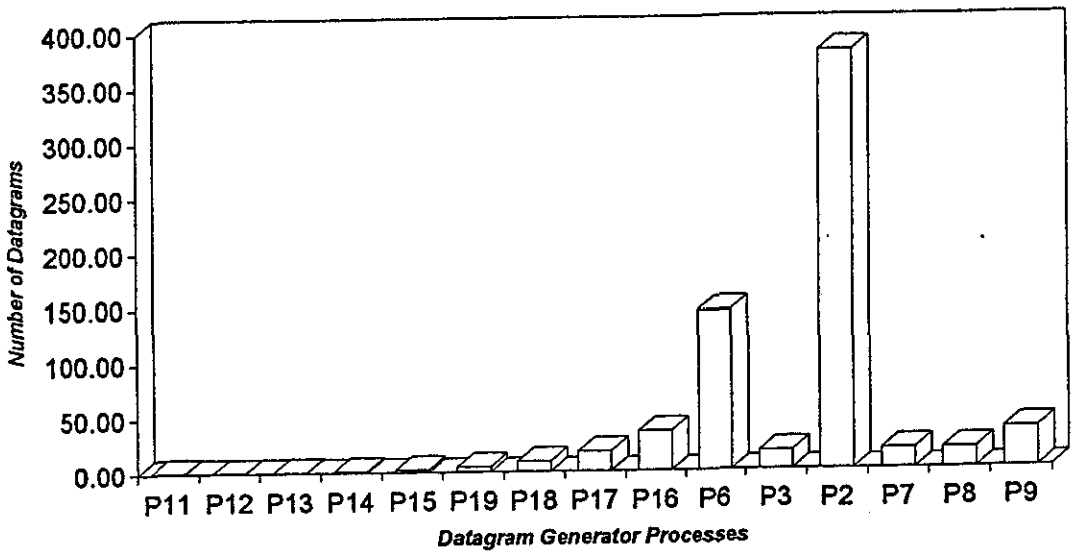


Figure 7.1 (14) : Distribution of datagrams originated from differing source nodes with varying distances to a fixed destination process **(Tracer)** under maximum communication load conditions.

The results are shown at **Figure 7.1(15)** and **Figure 7.1(16)**, for 3 and 4 processes, respectively. As can be seen in the figures, the distribution of datagrams to the nodes is almost balanced under normal communication conditions, regardless of the physical locations of the processes on the network.

In **Figure 7.1(16)**, the number of datagrams generated in the same time period is almost the same for **P11** (**Proc11** which is the farthest to the **Tracer**), **P19** (**Proc19** which is in the middle), and **P6** (**Proc6** which is one of the closest to the **Tracer**). In both cases (the test with 3 and 4 processes), **P2** (**Proc2**) could generate relatively more datagrams than others, because its node is the closest to the Tracer physically.

This can be considered as the worst case of a normal datagram traffic, and even in this extreme situation every node has an equal chance of sending messages to the destination regardless of its network location, guaranteeing the predictable delivery of a message in a maximum amount of time (the time required for the delivery of the farthest node's datagram) for any process.

7.2 Concluding Remarks

The failure rate of the Inmos Transputer link is sufficiently low to make checking for random errors superfluous. Therefore, the only realistic way to counter such disruptive intrusions as message corruption or loss, is to provide careful message integrity checking at the application level. Disruptive intrusions can be more easily handled at this level as counter measures can be constantly updated, while the minimal (micro) kernel must remain unchanged. This latter aspect has been the most important criteria of the RT-DOS IPC design, and an unreliable datagram service is the core of the IPC mechanism.

The RT-DOS IPC supports both reliable circuit switching service and unreliable message switching service as well as a wide range of group communication services namely broadcasting, domain casting, multicasting, and domain multicasting. All of these communication services are built on top of a simple but efficient unreliable datagram service which exploits the underlying network topology of the multi loop RT-DOS architecture.

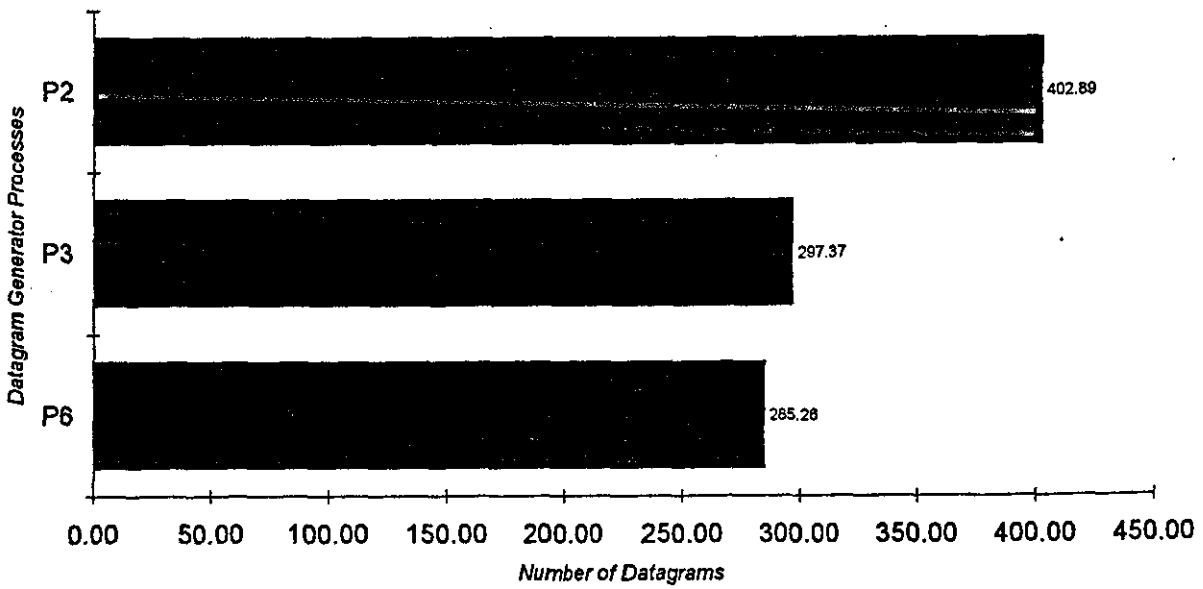


Figure 7.1 (15) : Distribution of datagrams reached to a fixed destination node (Tracer) from a number of source processes which reside on nodes with varying distances to the destination node.

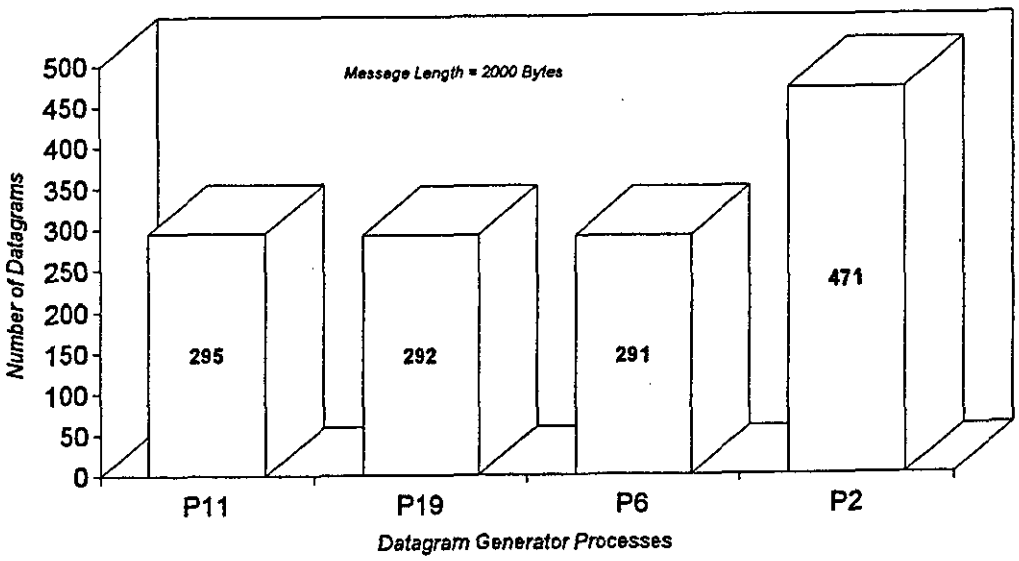


Figure 7.1 (16) : Distribution of datagrams originated from varying source nodes with different distances to a fixed destination process (Tracer) under the same communication load conditions.

At the beginning of the IPC design it was argued that using a datagram service for direct message transferring between processes would reduce the predictability of the RT-DOS communication services. I was, therefore, agreed that the datagram service should be used to establish connections only and messages (user data) should be transferred through physical links from one process space to another process space directly.

Nevertheless, this approach has been found to be very inefficient for group communication services as setting point-to-point connections between a sender process and all the recipients on the network for a broadcasting message would affect predictability of the overall system negatively, by bringing the physical limitations of the system down considerably, in comparison with the message passing protocol for exchanging the same number of messages through the network. In addition to this, without using the datagram service for message passing between processes it wouldn't be possible to exploit the advantages of the RT-DOS architecture which is based on multiloop topology; because messages that are restricted to only a specific application domain would be transferred through point-to-point connections even though the sender and receiver processes are physically on neighboring nodes within the domain.

In the currently adapted schema point-to-point connections are used to transfer bulk amount of data (more than 2KB) on user request, as it is becoming disadvantageous to use the datagram service for messages of longer than 2000 bytes. The RT-DOS kernel can differentiate between the long and the short messages from user communication requests, and decide which service is more efficient to use for delivering the message if the user explicitly state the service to be used. The upper limit for the length of messages to be transmitted through the IPC datagram service can be changed dynamically, as it would make sense because each network configuration would impose a different limits depending on the performance of available hardware components (processing power of Transputers, speed of links, and dynamic link switch performance, etc.). Currently, none of the protocols are imposed on the upper layers and they are free to choose any communication protocol and related policy to satisfy their own specific requirements: group communication, point-to-point connection, reliable blocked communication, or unreliable unblocked no-reply type of message exchange.

It is shown that with a proper network topology and system architecture, it is possible to implement group communication protocols with Transputer hardware (despite its architectural shortcomings), as well as synchronous point-to-point circuit switching protocols.

8 FURTHER RESEARCH AREAS

The targeted RT-DOS Kernel is not intended to be a marketable product, nor a released and supported facility for a community of application programmers. Rather, it is intended to be a research testbed, whose primary clients will be a small number of highly qualified system programmers and researchers performing experiments with operating system concepts and techniques for reliable decentralized real-time control systems.

In addition to the operating system field, the following areas can be investigated using the testbed, the IPC mechanism which has been implemented, and the other kernel primitives available :

- Implementation of concurrent programming languages;
- Study of parallel algorithms;
- Study of dynamic scheduling and load sharing on computer networks;
- Study of fast communication protocols on reliable media;
- Various real-time command and control application (robotics, switching, etc.).

APPENDICES

List of Figures

Glossary

IPC	: Interprocess Communication.
RPC	: Remote Procedure Call.
RT-DOS	: Real-Time Distributed Operating System.
OOP	: Object Oriented Programming.
OSI	: Open Systems Interconnection.
ISO	: International Standards Organization.
NOS	: Network Operating Systems.
RT	: Real-Time.
DOS	: Distributed Operating System.
OS	: Operating System.
Gbps	: Giga bits per second.
Mips	: Million instructions per second.
Mbps	: Mega bits per second.

References

[Accetta 1986] Accetta, M., Baron, R., Rashid, R., Mach: A new Kernel Foundation for Unix Development, In Proceedings of the Summer Usenix Conference, Atlanta, GA, July 1986.

[ArtsyA 1987] Artsy, Y., et al., Interprocess Communication in Charlotte, IEEE Software, Jan. 1987.

[ArtsyB 1989] Artsy, Y., Finkel, R., Designing a Process Migration Facility: The Charlotte Experience, IEEE Computer, Vol 22, No 9, Sept. 1989.

[Aytac 1992] Aytac, K., Tayli, M., Bor, M. Simulation Experimentation on Real-Time Distributed Systems. KSU-CCIS Research Center Project RP #36/407-408, March 1992.

[Balin 1989] Balin, S.C., An Object-Oriented Requirements Specification Method, ACM Comm., May 1989.

[Ball 1976] Ball, J., Feldman, J., RIG Rochester's Intelligent Gateway : System Overview, IEEE Transactions in Software Engineering (2,4), Dec. 1976.

[Baskett 1977] Baskett, F., et al., Task Communication in Demos, Proc. 6th ACM Symp. OS Princ., ACM Press, New York, Dec. 1977.

[Benmaiza 1990] Benmaiza, M., Tayli, M., Bor, M. Predictable Architecture for a Transputer-Based RT-DOS. In Proceedings of the SCS 12th National Computer Conference, Oct. 1990.

[Benmaiza 1993] Benmaiza, M., Tayli, WoTUG Proceedings, Circuit-switched IPC for Predictable Message Passing in a Multiloop Transputer Network, Germany, 22-23 September 1993.

[**Bihari 1992**] Bihari, E.T., Gopinant, P., Object-Oriented Real-Time Systems: Concepts and Examples, IEEE Computer, Dec.1992.

[**BorA 1989**] Bor, M., Tayli, M. Object-Oriented Approach to the Design of Real Time Distributed Operating Systems. KSU-CCIS Research Center Project RP #27/407-408, Dec. 1989.

[**BorB 1990**] Bor, M., Tayli, M. Object Oriented Approach to the Design of RT-DOS Systems. In Proceedings of the SCS 12th National Computer Conference, Oct. 1990.

[**Briat 1990**] Briat, J., at el., PARX: A Parallel Operating System for Transputer Based Machines, IMAG-LGI Laboratory, Univ. of Grenoble-BP. 53X F-38041, Grenoble, 1990.

[**BYTE 1986**] BYTE, Elements of Object-Oriented Programming, Aug. 1986.

[**CheritonA 1988**] Cheriton, D., The V Distributed System, Comm. ACM, Vol 31, No 3, March 1988.

[**CheritonB 1983**] Cheriton, D., Zwaenepoel, W., The Distributed V Kernel and its Performance for Diskless Workstations, In Proceedings of the 9th ACM Symposium on OS Principles (Bretton Woods, New Hampshire, Oct. 10-13), ACM, New York, 1983.

[**CheritonC 1986**] Cheriton, D., VMTP a Transport Protocol for the Next Generation of Communication Systems, In Proceedings of SIGCOMM 86, ACM, New York, Aug. 1986.

[**Cox 1986**] Cox, B.J., Object-Oriented Programming, Addison-Wesley, Reading, Massachusetts, 1986.

[**Danfort 1988**] Danfort, S., Tomlinson, C., Type Theories and Object-Oriented Programming, ACM Comp.Surveys, Vol 20, No 1, March 1988.

[**Datter1985**] Datter, R., OCCAM and the Transputer, Electronics & Power, April 1985.

[**Davson 1989**] Davson, J., Can Object-Oriented Databases be as successful as relational databases ?, BYTE, Sept 1989.

[**Dijkstra 1968**] Dijkstra, E.W., The Structure of the THE Multiprogramming System, Comm. of the ACM, Vol 11, May 1968.

[**Eckhouse 1989**] Dyke, R.P., Kunz, J.C., Object-Oriented Programming, IBM Sys. Journal, Vol 28, No 3, 1989.

[**Eckhouse 1986**] Eager, D., Lazowska, E., Adaptive Load Sharing in Homogeneous Distributed Systems, IEEE Transactions in SE, May 1986.

[**Eckhouse 1978**] Eckhouse, R.H.Jr., Stankovic, J.A., Issues in Distributed Processing-An Overview of two Workshops. IEEE, Computer, Vol 11, No 1, 1978.

[**Enslow 1978**] Enslow, P.H.Jr. What is a Distributed Data Processing System ?, IEEE Computer, Vol 11, No 1, 1978.

[**Gaughan 1993**]Gaughan, P.T., Yalamanchili, S., Adaptive Routing Protocols for Hypercube Interconnection Networks, IEEE Computer, May 1993.

[**Geraint 1987**] Geraint, J., Programming in Occam, Prentice-Hall, U.K., 1987.

[**Goldberg 984**] Goldberg, A., Smalltalk-80 : The Language and Its Implementation, MA:Addison-Wesley, 1984.

[**Grimsdale 1989**] Grimsdale, C.H.R., Distributed Operating System for Transputers, Microprocessors and Microsystems, Vol 13, No 2, March 1989.

[**Grimshaw 1993**] Grimshaw, A.S., Easy-to-Use Object-Oriented Parallel Programming with Mentat, IEEE Computer, May 1993.

[**Haberman 1970**] Haberman, A.N., Flon, L., Modularization and Hierarchy in a Family of Operating Systems, Communications of the ACM 19(5):266-272, May 1970.

[**Hansen 1970**] Hansen, B.P., The Nucleus of a Multiprogramming System, Communications of the ACM 13(4):238-250, April 1970.

[**Hariri 1992**] Hariri, S., Sarikaya, B., Architectural Support for Designing Fault-Tolerant Open Distributet Systems, IEEE Computer, Vol 25, No 6, June 1992.

[**Helios 1989**] Helios, Periphelion Software Ltd., The HELIOS Operating System, Prentice Hall, ISBN 0-13-386004-3, 1989.

[**HideyukiA 1989**] Hideyuki, T., Clifford, W. ARTS : A Distributed Real-Time Kernel, ACM Operating Systems Review, Vol 23, No 3, July 1989.

[**HideyukiB 1987**] Hideyuki, T., Mercer, C.W., ARTS : A Distributed Real-Time Kernel, CMU, CS Dept., Pittsburg, Pennsylvania, 1987.

[**Hoare 1978**] Hoare, C.A.R. Communicating Sequential Processes, Communication of ACM, Vol 21, No 8, 1978.

[**Hull 1989**] Hull, M.E., Alibadi, A.Z., Real-Time System Implementation-The Transputer and OCCAM Alternative, Microprocessing and Microcomputing 26, 1989.

[**Hutchisson 1989**] Hutchisson, N.C., et al., RPC in the x-Kernel: Evaluating New Design Techniques, communication of ACM, March 1989.

[**HYDRA/OS 975**] HYDRA/OS, OS Principles, ACM, Overview of the HYDRA/OS, in Proc. 5th Symp., Nov. 1975.

[InmosA1987] Inmos, INMOS Ltd., Spectrum-Production Brochure, 42-0180-00, March 1987.

[InmosB 1987] Inmos, INMOS Ltd., IMS T414 Transputer Preliminary Data, 42-1078-01, Feb. 1987.

[InmosC 1989] Inmos, Transputer Technical Notes. ISBN 0-13-929126-1 Prentice-Hall, 1989.

[InmosD 1989] Inmos, The Transputer Databook, INMOS document number 72 TRN 203 01, Redwood Burn Ldt, Trowbridge, 1989.

[InmosE 1987] Inmos, INMOS Ltd, The Transputer Instruction Set - a Compiler Writer's Guide, 77-OCC-04600, March 1987.

[InmosF 1988] Inmos, INMOS Ltd., OCCAM 2 Reference Manual (Prentice Hall), 1988.

[InmosG 1990] Inmos, INMOS Ltd, ANSI C Toolset User Manual, 72 TDS 224 00, Aug. 1990.

[InmosH 1990] Inmos, INMOS Ltd, Transputer Development System (Second Edition), Prentice Hall, ISBN 0-13-929068-0, London 1990.

[InmosI 1988] Inmos, INMOS Ltd, Communicating Process Architecture, ISBN 0-13-629320-4 Prentice-Hall, 1988.

[InmosJ 1991] Inmos, INMOS Ltd., Preliminary Data IMS C004 Programmable Link Switch, Bristol, UK, 1991.

[Jones 1979] Jones, A.K. et al., StarOS, a Multiprocessor Operating System for the Support of Task Forces, Proceedings of the 7th ACM Symposium on OS Principles, 1979.

[**Kaashoek 1990**] Kaashoek, M.F., Tanenbaum, S.F., An Efficient Reliable Broadcast Protocol, Dept. of Math. & CS, Vrije University, Amsterdam, The Netherlands, 1990.

[**Khan 1981**] Khan, K.C. & at el, iMAX : A Multiprocessing Operating System for an Object-Based Computer, Communications ACM , Dec. 1981.

[**Kimbleton 1978**] Kimbleton, S.R. et al., Network Operating System- An Implementation Approach, Proceedings of NCC, Vol 47, 1978.

[**Kohler 1981**] Kohler, W.H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, Comp.Surveys, Vol 13, No 12, June 1981.

[**Lamberts 1993**] Lamberts, S., Parallel Processing in Germany, Newsletter of the Parallel Processing Technical Committee, January 1993.

[**Leland 1986**] Leland, W., Ott, T., Load Balancing Heuristics and Process Behavior, In proceedings ACM OS Review (pp 54-69), July 1986.

[**Livny 1982**] Livny, M., Melman, M., Load Balancing in Homogeneous Broadcast Distributed Systems, In Proceedings of the Computer Networks Performance Symposium (pp 47-55), April 1982.

[**Lo 1989**] Lo, V., Process Migration for Communication Performance, IEEE OS Tech. Comm. Newsletter (3,1), Winter 1989.

[**Lugue 1993**] Lugue, E. et al., Distributed Kernel for a Transputer Baser Computer, Transputer Applications and Systems, IOS Press, 1993.

[**May 1986**] May, D., Shepherd, R. The Transputer Implementation of Occam, Esprit Summer School on Future Parallel Computers, 1986.

[**Mayer 1988**] Mayer, B., Object-Oriented Software Construction, NJ:Prentice-Hall, 1988

[McQuillan 1977] McQuillan, J.M., Walden, D.C., The ARPA Network Design Decisions, Computer Networks, Vol 1, No 3, 1977.

[Milonkovic 1992] Milonkovic, M., Operating Systems : Concepts and Design, McGraw-Hill, Inc., Singapore, ISBN NO : 0-07-112711-9, 1992.

[Mockapetris 1983] Mockapetris, P.V., Analysis of Reliable Multicast Algorithms for Local Networks, Proc. 8th Data Comm. Symp., pp 150-157, Silver Spring, MD, Oct. 1983.

[Murray 1988] Murray, K.A., Wellings, A.J., Issues in the Design and Implementation of a Distributed Operating System for a Network of Transputers, Microprocessing and Microprogramming, 24 (1988) 169-178.

[Needham 1982] Needham, R.M., Herbert, A.J., The Cambridge Distributed Computing System, Addison-Wesley, Reading, 1982.

[Newman 1982] Newman, I.A., The Organization and use of Parallel Processing Systems (143-159), Parallel Processing Systems, Edited By Prof.Dr. D.J. Evans, Combridge University Press, 1982.

[Ni 1993] Ni, L.M., McKinley, P.K., A Survey of Wormhole Routing Techniques in Direct Networks, IEEE Computer, February 1993.

[Nortcutt 1987] Nortcutt, J.D., Mechanisms for Reliable Distributed Real-Time Operating Systems : The ALPHA Kernel, Academic Press Inc., 1987.

[Oakley 1989] Oakley, H., Mercury : an Operating System for Medium-grained Parallelism, Microprocessors and Microsystems, Vol 13, No 2, March 1989.

[Ousterhout 1980] Ousterhout, K.J., et al., MEDUSA: An Experiment in Distributed Operating System Structure, Communications of ACM, Vol 23, No 2, 1980.

[**Polymorphism 1985**]Polymorphism, On Understanding Types, Data Abstraction, and Polymorphism, Comp. Surveys, Vol. 17, No 4, Dec. 1985.

[**Pountain 1987**] Pountain, D., A Tutorial Introduction to OCCAM Programming, Inmos, 72-TRN-11903, June 1987.

[**Randell 1983**] Powell, M., Miller, B., Process Migration in DEMOS/MP, In Proceedings of the 9th ACM Sym. on OS Principles, ACM, New York, Oct. 1983.

[**Randell 1978**] Randell, B., et al., Reliability issues in Computing System Design, Computing Surveys, Vol 10, No 2, 1978.

[**RashidA 1981**] Rashid, R.F., et al., Accent: A communication Oriented Network Operating System Kernel, Proc. 8th ACM Symp. OS Princ., ACM Press, New York, Dec. 1981.

[**RashidB 1981**] Rashid, R.R., ACCENT : A communication Oriented NOS Kernel, Department of CS, CMU, April 1981.

[**Renterghem 1989**] Renterghem, P.V., Transputer for Industrial Applications, Concurrency: Practice and Experience, Vol 1(2), 135-169, Dec. 1989.

[**Sakamura 1987**] Sakamura, K. The TRON Project, IEEE Micro, Vol 7, No 2, 1987.

[**Shipman 1987**] Shipman, S.E., Mechanisms to Support Object Replication for a Distributed Kernel, Ph.D. thesis, CSC Dept., CMU, 1987.

[**StankovicA 1984**] Stankovic, J.A. A perspective on Distributed Computer Systems, IEEE Transaction on Computers, Vol c-33, No 12, 1984.

[**StankovicB 1989**] Stankovic, J.A., Ramamritham, K., The Spring Kernel : A new Paradigm for Real-Time Operating Systems, ACM Operating Systems Review, Vol 23, No 3, July 1989.

[StankovicC 1985] Stankovic, J.A., Real-Time Computing : The Next Generation, IEEE Transactions on Computer, Vol 34, No 12, 1985.

[StankovicD 1985] Stankovic, J.A., et al. A Review of Current Research and Critical Issues in Distributed System Software, IEEE Distributed Processing Technical Committee Newsletter, Vol 7, No 1, 1985.

[StankovicE 1988] Stankovic, J.A., Misconceptions About Real-Time Computing, IEEE Computer, October 1988.

[StankovicF 1988] Stankovic, J.A., A Serious Problem for Next Generation Systems, IEEE Computer, Oct. 1988.

[Stroustrup 1986] Stroustrup, B., The C++ Programming Language, Addison-Wesley, 1986.

[T9000 1992] T9000, WoTUG Newsletter, T9000Systems Workshop, No 17, July 1992.

[TanenbaumA 1990] Tanenbaum, A.S., et al., Experiences with the AMOEBA Distributed Operating System, Communion of ACM, Vol 33, No 12, Dec. 1990.

[TanenbaumB 1987] Tanenbaum, A.S., Operating Systems: Design and Implementation, Prentice-Hall International, Inc., 1987.

[TanenbaumC 1981] Tanenbaum, A.S., et al., An Overview of the Amoeba DOS, OS Review , Vol 15, No 3, July 1981.

[TanenbaumD 985] Tanenbaum, A.S., et al., Distributed Operating Systems, ACM Computing Surveys, Vol 17, No 4, Dec. 1985.

[TanenboumE 1981] Tanenboum, A.S., Computer Networks, Prentice/Hall International, 1981.

[Tay 1990] Tay, B.H., Ananda, A.L., A Survey of Remote Procedure Calls, Dept. of IS & CS, National University of Singapore, Kent Ridge Crescent, Singapore 0511, 1990.

[TayliA 1987] Tayli, M., Eskicioglu, R., Bor, M. A Real Time Distributed Operating System Kernel For Industrial Applications. KSU-CCIS Research Center Project RP #27/407-408, Sept. 1987.

[TayliB 1990] Tayli, M., R., Bor, M. A Real Time Distributed Operating System Kernel For Industrial Applications-Preliminary Design Document. KSU-CCIS Research Center Project RP #27/407-408, January 1990.

[TayliD 1986] Tayli, M., Eskicioglu, R. Distributed Computer System Concepts & Literature Survey. KSU-CCIS Research Center Project RP #27/407-408, March 1986.

[TayliD 1990] Tayli, M., Bor, M., Benmaiza, M. A RT-DOS Kernel for Transputer-Based Systems. In Proceedings of the SCS 12th National Computer Conference, Oct. 1990.

[TayliE 1990] Tayli, M., Bor, M., Eskicioglu, R. RT-DOS A Real-Time Distributed Operating System Kernel for Transputers. In Proceedings of the OUG 13, 13th OCCAM User Group Technical Meeting on Real-Time Systems with Transputers, York, UK, Sept. 1990.

[TayliF 1989] Tayli, M, et al., Application Management System for a Real-Time Distributed Operating System, KSU - CCIS Research Project Proposal, Riyadh,/S.Arabia, Nov. 1989.

[Transputer 1984]Transputer, "Transputer - a programmable component that gives micros a new name", Computer Design, Feb. 1984.

[TranstechA 1993] Transtech, Parallel System Ltd., TTM60-C40 TRAM, Transtech Times, pp 5-6, Summer 1993

[TranstechB 1993] Transtech, Parallel System Ltd., C40s and Transputer Talk, 3LThreads, p.3, Summer 1993.

[Turek 1992] Turek, J., Shasha, D., The many Faces of Consensus in Distributed Systems, IEEE Computer, June 1992.

[Tyrrell 1989] Tyrrell, A.M., Nicoud, J.D., Scheduling and Parallel Operations on the Transputer, Microprocessing and Microprogramming 26, 175-185, 1989.

[Udell 1989] Udell, J., Clash of the Object-Oriented Pascals, BYTE July 1989.

[Vetter 1993] Vetter, R.J., Distributed Computing with High-Speed Optical Networks, IEEE Computer, February 1993.

[Walker 1983] Walker, B., et al., The LOCUS Distributed Operating System, In Proceedings of the 9th ACM Sym. on OS Principles, ACM, New York, Oct. 1983.

[Waring 1990] Waring, L.C., A general Purpose Communication Shell for a Network of Transputers, Microprocessing and Microcomputing, 29(1990) 107-119, Nort Holland, 1990.

[Watson 1988] Watson, R., Distributed System Architecture and Implementation : Identifiers in Distributed Systems, Springer-Verlag, New York, 1988.

[Wexler 1989] Wexler, J., Prior, D., Solving Problems with Transputers : Background and Experience, Microprocessors and Microsystems, Vol 13, No 2, March 1989.

[Wiederhold 1986] Wiederhold, G., Views, Objects, and Databases, IEEE Comp., Dec. 1986.

[**Woodward 1981**] Woodward, M.C., Some Aspects of the Efficient Use of Multiprocessor Control Sysyems, Ph.D. Thesis, Loughborough University of Technology, UK, 1981.

[**Wulf 1981**] Wulf, W.A., et al., HYDRA/C.mmp: An Experimental Computer System, McGraw-Hill, New York, 1981.

[**Zimmerman 1980**] Zimmerman, H. OSI Reference Model - The ISO Model Architecture for Open Systems Interconnection, IEEE Transactions on Communications, Vol 28, No 4, 1980.